UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Ernest Štukelj

Razvoj jezika za iskanje, povezovanje in predstavitev podatkov

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU
SMER PROGRAMSKA OPREMA

Ljubljana, 2016

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Ernest Štukelj

Razvoj jezika za iskanje, povezovanje in predstavitev podatkov

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU
SMER PROGRAMSKA OPREMA

MENTOR: doc. dr. Tomaž Dobravec

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Razvoj jezika za iskanje, povezovanje in predstavitev podatkov

Tematika naloge:

V diplomskem delu preglejte področje poizvedovalnih jezikov. Izberite nekaj pogosto uporabljenih predstavnikov, jih preučite in v delu opišite. Izbrane jezike med seboj primerjajte po različnih kriterijih.

V delu preučite tudi poizvedovalni jezik, ki ga uporablja sistem ALGator. Ugotovite šibke točke obstoječega jezika in predlagajte možne izboljšave. V sistemu ALGator implementirajte svoj poizvedovalni jezik, ki bo omogočal sestavljene poizvedbe, in ga primerjajte z obstoječim jezikom.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Ernest Štukelj,

sem avtor diplomskega dela z naslovom:

Razvoj jezika za iskanje, povezovanje in predstavitev podatkov

S svojim podpisom zagotavljam, da:

- sem diplomsko nalogo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 25.8.2016 Ernest Štukelj

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
LINQ	Language-integrated query	Jezikovno integriran poizvedovalni jezik
SQL	Structured query language	Strukturiran poizvedovalni jezik
JSON	JavaScript Object Notation	Objektni zapis v jeziku JavaScript
XML	Extensible Markup Language	Razširljiv označevalni jezik

Povzetek

Naslov: Razvoj poizvedovalnega jezika za iskanje, povezovanje in predstavitve podatkov

Cilj diplomske naloge je bil izboljšanje sistema Algator v delu, ki se ukvarja z analizo in predstavitvijo podatkov. Najprej smo primerjali nekaj poizvedovalnih jezikov, da bi našli čim bolj primernega za uporabo v sistemu Algator. Algator shranjuje podatke v točno določeno obliko, podatki pa se združujejo vedno na enak način, zato je lasten poizvedovalni jezik bolj primeren, ker lahko upošteva te lastnosti. Določili smo poenostavljeno sintakso SQL in jo uporabili v sistemu Algator. Za pisanje poizvedb je bil razvit uporabniški vmesnik in pretvornik poizvedb iz formata JSON v poenostavljen SQL ter obratno. Izvedba poizvedb je bila optimizirana, ker so nekatere poizvedbe trajale predolgo. Dodana je bila funkcionalnost izračunanih polj, ki omogočajo, da v rezultat poizvedb dodamo še polja, ki so izračunana iz ostalih polj.

Ključne besede: Algator, poizvedovalni jeziki, SQL, LINQ, XQuery, optimizacija.

Abstract

Title: Development of query language for retrieval, integration and presentation of data

The aim of this work was to improve the system Algator in the part, which deals with the analysis and presentation of data. First, we compared some query languages to find the most suitable for use in the system Algator. Algator stores data in a specific format, and the data are always combined in the same fashion. Therefore, special query language is more suitable because it can take advantage of these properties. We determined the simplified syntax of SQL and used in the system Algator. User interface and converter from JSON format to a simplified SQL and vice versa has been developed. Execution of queries has been optimized because some inquiries run too long. Functionality of calculated fields has been added, which allows to add additional fields that are calculated from other fields.

Keywords: Algator, query languages, SQL, LINQ, XQuery, optimization.

Poglavje 1 Uvod

Sistem Algator je namenjen izvajanju algoritmov na podanih testnih podatkih ter analizo rezultatov izvajanja. Rezultate izvajanja shranjuje v tabele, nad katerimi se potem lahko izvajajo poizvedbe. Poizvedbe se urejajo preko uporabniškega vmesnika ali pa direktno v lastnem formatu v obliki JSON. Algator tabele združuje vedno na enak način, ve se kateri stolpci so ključni podatki, zato splošen poizvedovalni jezik ni potreben. Izvedba poizvedb ni optimalna, manjka pa tudi kakšna funkcionalnost. Cilj diplomske naloge je dopolnitev dela sistema, ki se ukvarja z analizo in predstavitvijo podatkov. V sistem smo hoteli dodati poizvedovalni jezik, ki bi bil morda bolj primeren za programiranje in bi omogočal še kakšne dodatne funkcionalnosti.

Najprej smo primerjali več obstoječih poizvedovalnih jezikov po različnih lastnostih. Za primerjavo smo izbrali SQL, LINQ ter XQuery. Primerjali smo podatkovne modele, sintakso, funkcionalnost in nekaj splošnih lastnosti programskih jezikov. Namen vsakega od treh poizvedovalnih jezikov je malo drugačen. SQL je namenjen poizvedovanju nad relacijskimi bazami, in na tem področju ima veliko funkcionalnosti. LINQ je integriran v drug programski jezik, kar nam omogoča izdelavo celotne aplikacije v enem jeziku, ne glede na vrste virov podatkov, ki jih uporabljamo. XQuery je pa namenjen poizvedovanju nad dokumenti XML.

Obstoječa sintaksa poizvedbe je v formatu JSON (slika 1.1). V poizvedbi naštejemo algoritme za katere nas zanimajo rezultati, testne množice, vhodne in izhodne parametre ter filter, sortiranje in grupiranje. S `ComputerID` izberemo nabor tabel. Na testne množice lahko gledamo kot na tabele nad katerimi se bo izvedla poizvedba. Z izbranimi algoritmi in parametri izberemo stolpce v teh tabelah, ki nas zanimajo. S filtrom in grupiranjem omejimo še vrstice.

```
{"Query": {
  "Name": "?",
  "Algorithms": [
    "JJava7 AS JJava7",
    "CJava7 AS CJava7"
  ],
  "TestSets": [
    "TestSet0 AS TestSet0",
    "TestSet1 AS TestSet1"
  ],
  "Parameters": [],
  "Indicators": [
    "Tmax AS Tmax",
    "CMP AS CMP",
    "Tmax-Tmin AS Diff"
  ],
  "GroupBy": [""],
  "Filter": ["Tmax>20"],
  "SortBy": [""],
  "Count": "0",
  "ComputerID": "F1.C1"
}}
```

Slika 1.1: Primer poizvedbe v Algatorju v format JSON.

Poskusili smo določiti dodatno sintakso za poizvedovalni jezik, ki bi bil zadosten za uporabo v sistemu Algator, ter čim bolj enostaven za uporabo. Za ročno urejanje poizvedbe format JSON ni najbolj primeren, ker moramo paziti na narekovaje, pravilno rabo zavitih in oglatih oklepajev, vedeti moramo tudi ali je potrebno v določeno polje vnesti seznam vrednosti ali samo eno vrednost. Nova sintaksa je podobna jeziku SQL (slika 1.2), ne vsebuje pa vseh njegovih sklopov in možnosti, ker so podatki vedno v vnaprej znani obliki, različne tabele se pa vedno med seboj povezujejo na enak način. Nov jezik smo poimenovali A-SQL.

```
FROM TestSet0, TestSet1
WHERE (algorithm=JJava7 OR algorithm=CJava7)
AND Tmax>20
AND ComputerID=F1.C1
SELECT Tmax, CMP, Tmax-Tmin AS Diff
```

Slika 1.2: Primer poizvedbe v Algatorju v sintaksi A-SQL. Rezultat poizvedbe je na sliki 1.3, enak rezultat dobimo tudi s poizvedbo v formatu JSON iz slike 1.1.

Tudi rezultat poizvedbe v Algatorju (slika 1.3) ni takšen kot ga bi pričakovali, če bi gledali poizvedbo iz slike 1.2 iz vidika jezika SQL. V poizvedbi smo namreč našteli samo 3 stolpce, rezultat pa jih vsebuje več, za vsak algoritem po 3.

ID	Testset	TestID	Pass	JJava7.Tmax	CJava7.Tmax	JJava7.CMP	CJava7.CMP	JJava7.Diff	CJava7.Diff
2	TestSet0	Test-2	DONE	1982	38	266	?	1939	0
3	TestSet0	Test-3	DONE	455	50	682	?	318	0
4	TestSet0	Test-4	DONE	620	73	1053	?	411	0
5	TestSet0	Test-5	DONE	2503	85	1307	?	2098	0
6	TestSet0	Test-6	DONE	6381	130	1566	?	6115	0
7	TestSet0	Test-7	DONE	1201	132	2197	?	1021	0
8	TestSet0	Test-8	DONE	6673	154	2634	?	6462	0
9	TestSet0	Test-9	DONE	6276	176	3339	?	5994	0
10	TestSet0	Test-10	DONE	4468	188	3831	?	4144	0

Slika 1.3: Rezultat poizvedbe iz slike 1.1 ali 1.2.

Nov poizvedovalni jezik smo vgradili v sistem tako, da je njegovo izvajanje ostalo čim bolj enako prvotnemu. Prvotna izvedba sicer ni bila optimalna, nekatere poizvedbe so se izvajale predolgo, zato smo izvajanje optimizirali. Z orodjem Profiler smo določili kritične dele izvajanja programa ter jih v nekaj korakih spremenili na več načinov. Pri optimizaciji smo spremenili določene podatkovne strukture, da je postalo iskanje po njih bolj učinkovito. Spremenjen je bil tudi sam postopek izvajanja poizvedb. V prvotni izvedbi so se nekateri koraki izvajanja ponavljali tudi ko to ni bilo potrebno. Dodali smo še predpomnjenje rezultatov poizvedb, kar je pohitrilo izvajanje naslednjih podobnih poizvedb.

Dodali smo še funkcionalnost izračunljivih polj z uporabo interpreterja programske kode v Javi. To nam omogoča, da v sklopu `SELECT` (A-SQL) oz. `Indicators` (JSON) v programskem jeziku Java zapišemo izraz sestavljen iz parametrov, operatorjev, konstant ter funkcij jezika Java. Izraz se pri izvedbi poizvedbe izračuna s konkretnimi vrednostmi parametrov ter prikaže v rezultatu kot stolpec.

Poglavje 2 Primerjava poizvedovalnih jezikov

2.1 SQL

SQL je strukturirani poizvedovalni jezik. Baziran je na relacijski algebri in relacijskem računu. Vsebuje jezik za definiranje, manipulacijo in nadzor podatkov. Uporablja relacijski podatkovni model. Jezik je deklarativen, vsebuje pa tudi proceduralne elemente. Vsebina poglavja je povzeta po [4].

2.1.1 Podatkovni model

SQL uporablja relacijski podatkovni model, ki je sestavljen iz zbirke relacij. Relacijo si lahko predstavljamo kot tabelo. Vrstica tabele se v relacijskem podatkovnem modelu imenuje n-terica, ime stolpca pa atribut. Podatkovni tip atributa se imenuje domena. Domena je množica osnovnih vrednosti. Osnovna vrednost je nedeljiva v podatkovnem modelu. Domena je definirana z imenom, podatkovnim tipom in formatom. Podatkovni tipi so npr. število, niz, datum, itd., format je pa dodatna informacija o osnovnih vrednosti, ki pripadajo domeni. Če bi želeli npr. definirati domeno slovenskih mobilnih telefonskih številk, bi izbrali podatkovni tip niz, format bi bil pa oblike »nnn nnn nnn«, kjer je n število od 0 do 9.

Relacijska shema je sestavljena iz imena relacije in seznamom atributov. Vsak atribut ima določeno domeno. Število atributov imenujemo stopnja relacije.

Relacija oz. stanje relacije je množica n-teric. Te so v relaciji definirane kot množica, kar pomeni, da nimajo določenega vrstnega reda. Predstavimo jih lahko tudi kot množico parov »atribut« in »vrednost«, kar pomeni, da točen vrstni red atributov ni pomemben, lahko se tudi spremeni, mora se pa ohranit povezava z njegovo vrednostjo.

Vsako vrednost v n-terici je osnovna, kar pomeni, da ni deljiva. Nek atribut lahko tudi nima določene vrednosti, v tem primeru ima ničelno vrednost (ang. `null`).

Relacija sestavljena iz množice n-teric, kar po definiciji množice pomeni, da v množici n-teric ne obstajata dve enaki n-terici. Dve n-terici ne moreta imeti enakih vseh parov `<atribut>,<vrednost>`. Po navadi obstaja neka podmnožica atributov, ki že zadošča, da dve n-terici zagotovo nista enaki. Takšno podmnožico atributov imenujemo nadključ relacije. Vsaka relacija ima vsaj en nadključ – celotno množico atributov. Minimalni nadključ imenujemo ključ. Pomeni, da je ključ minimalna množica atributov, ki zadošča, da ne obstajata dve n-terici z enakimi vrednostmi atributov, ki so v tem ključu. Takšnih minimalnih

množic je v relaciji lahko tudi več, tako da takšne množice imenujemo kandidati za ključ. Izmed kandidatov za ključ po navadi izberemo en ključ, ki postane primarni ključ relacije. Zaželeno je, da je v primarnem ključu čim manj atributov. Atributi v primarnem ključu ne morejo imeti ničelne vrednosti.

Del relacijskega podatkovnega modela so tudi celovitost (ang. integrity), sklicevalna celovitost (ang. referential integrity) in tuji ključi (ang. foreign keys)

Sklicevalna celovitost je določena med dvema relacijama. Če je določena, zahteva, da se n-terica v neki relaciji mora sklicevati na n-terico v drugi relaciji. Množico atributov v prvi relaciji, ki ustreza primarnemu ključu v drugi relaciji, imenujemo tuji ključ.

2.1.2 Definiranje podatkov

SQL uporablja izraze tabela, vrstica in stolpec za izraze relacija, n-terica in atribut iz relacijskega podatkovnega modela.

Jezik za definiranje podatkov vsebuje ukaze za ustvarjanje (slika 2.1), spreminjanje in brisanje shem, tabel, atributov, domen (slika 2.2) in omejitev. Vrhnji element v podatkovni bazi SQL je katalog, ki vsebuje več shem, sheme pa vsebujejo relacije oz. tabele.

```
CREATE TABLE T_KOPIJA (stevilka varchar(50) NOT NULL,
    kopija int NOT NULL,
    datum datetime NOT NULL,
    vnasalec varchar(50) NOT NULL,
 CONSTRAINT [PK_T_KOPIJA] PRIMARY KEY CLUSTERED
(
    stevilka ASC,
    kopija ASC
))
```

Slika 2.1: Primer ukaza za ustvarjanje nove tabele.

SQL podpira veliko podatkovnih tipov, delimo jih na numerične, črkovne, niz bitov, logične in datumske. Možno pa je definirati tudi domene in razne omejitve. Domene so uporabne, če želimo nekemu podatkovnemu tipu dodeliti svoje ime. Na ta način lahko potem enostavno spremenimo ta podatkovni tip atributom po celotni bazi, ker samo z enim ukazom spremenimo domeno. Omejitve nam omogočajo, da nek atribut ali domeno še dodatno omejimo znotraj podatkovnega tipa. Če ima nek atribut na primer določeno numerični podatkovni tip, lahko dodamo omejitev, da mora biti vrednost atributa znotraj nekega intervala.

```
CREATE DOMAIN starost AS INTEGER CHECK (starost>=0 AND starost<=150)
```

Slika 2.2: Primer definiranja domene z omejitvijo.

2.1.3 Manipulacija podatkov

2.1.3.1 Stavek SELECT

Osnovna izjava za pridobivanje podatkov je izjava `SELECT`. Sestavljena je iz naslednjih sklopov:

- `SELECT`: predstavlja projekcijo. Podati moramo seznam atributov, ki naj bodo v rezultatu poizvedbe. Če hočemo pridobiti vse atribute, vpišemo zvezdico (`*`).
- `FROM`: tukaj naštejemo relacije oz. tabele iz katerih se naj črpajo podatki.
- `WHERE`: opišemo pogoj oziroma filter. Ta sklop ni obvezna.
- `ORDER BY`: opišemo na kakšen način naj bo rezultat urejen.

```
SELECT *
FROM T_KOPIJA
WHERE datum<GETDATE()
ORDER BY datum
```

	stevilka	kopija	datum	vnasalec
1	150000000031	0	2015-11-23 11:09:29.100	1
2	150000000032	0	2015-11-23 11:10:27.917	1
3	150000000033	0	2015-11-23 13:29:28.163	1
4	150000000034	0	2015-11-23 13:30:26.907	1
5	150000000034	1	2015-11-23 14:15:37.803	1
6	150000000035	0	2015-11-23 14:52:06.897	1
7	150000000035	1	2015-12-01 11:31:52.327	1
8	150000000036	0	2015-12-01 11:37:20.793	1

Slika 2.3: Enostavna poizvedba v jeziku SQL in rezultat.

Podatke lahko tudi združujemo po določenem atributu, ali večih atributih. To pomeni, da hočemo v eno vrstico združiti vrstice, ki imajo določen atribut enak. Ker bo v tem primeru iz večih vrstic nastala ena, moramo rezultatu določiti vrednost katere vrstice naj bo v ostalih atributih, po katerih ne bomo združevali (slika 2.4). To naredimo z uporabo ene od agregacijskih funkcij.

```
SELECT stevilka, MAX(kopija)
FROM T_KOPIJA
WHERE datum<GETDATE()
GROUP BY stevilka
ORDER BY stevilka
```

	stevilka	kopija
1	150000000004	1
2	150000000009	3
3	150000000015	2
4	150000000026	1
5	150000000029	2
6	150000000031	1
7	150000000032	1
8	150000000033	1

Slika 2.4: Primer poizvedbe z združevanjem oz. grupiranjem in rezultat. Ker smo združevali po številki, zanimala nas je pa najvišja kopija, smo v stavek `SELECT` zapisali številka in `Max(kopija)`.

2.1.3.2 Stavki INSERT, DELETE in UPDATE

S stavki `INSERT, DELETE in UPDATE` vstavljamo, brišemo in spreminjamo podatke v tabelah.

```
INSERT INTO T_KOPIJA(stevilka, kopija, datum, vnasalec)
VALUES ('123', 0, GETDATE(), 'janez')

INSERT INTO T_KOPIJA(stevilka, kopija, datum, vnasalec)
SELECT *
FROM A_KOPIJA
```

Slika 2.5: Dve obliki vstavljanja podatkov.

Podatke lahko vstavljamo na dva načina, z naštevanjem točnih vrednosti ali pa tabelo napolnimo z rezultatom, ki nam ga vrne poizvedba (slika 2.5). V vsakem primeru najprej napišemo `INSERT INTO` in ime tabele v katero želimo podatke vstaviti.

```
DELETE FROM T_KOPIJA WHERE stevilka='123'
```

Slika 2.6: Primer izjave za brisanje podatkov iz tabele.

Za brisanje podatkov uporabimo izjavo `DELETE` (slika 2.6). Napišemo ime tabele iz katere hočemo brisati podatke ter filter, katere podatke hočemo izbrisati. Če sklop `WHERE`

izpustimo, bomo izbrisali vse podatke v tabeli. Če hočemo biti prepričani, da bomo izbrisali natanko en zapis, moramo v filtru zajeti celoten primarni ključ tabele, ali pa kater drug atribut, ki ima unikatno vrednost v tabeli.

```
UPDATE T_KOPIJA
SET kopija=3
WHERE stevilka='123' AND kopija=2
```

Slika 2.7: Primer izjave za spreminjanje podatka v tabeli.

Za spremembo nekega podatka v tabeli uporabimo izjavo `UPDATE` (slika 2.7). Najprej določimo v kateri tabeli hočemo spremeniti vrednost atributa, potem napišemo kako se naj atribut spremeni, in na koncu še filter, s katerim določimo vrstice katerim se naj spremeni vrednost atributa.

2.1.4 Nadzor podatkov

SQL za nadzor nad podatki uporablja različne vloge in uporabnike podatkovne baze. Administrator podatkovne baze je uporabnik, ki ureja pravice nad bazo. Ima naslednje zadolžitve:

- Kreiranje uporabniških računov: nastavlja uporabniška imena, skupine in gesla.
- Podeljevanje privilegijev: določenim računom dodeljuje določene pravice.
- Odvzemanje privilegijev: privilegije lahko računom tudi prekliče.
- Določa nivoje pravic računom.

Privilegiji se lahko določajo na več načinov. Lahko se določijo uporabniškemu računu, lahko pa tudi posamezni tabeli določimo kdo ima pravico do vpogleda ali spreminjanja podatkov.

2.1.5 Dodatne funkcionalnosti jezika SQL

SQL ima še veliko funkcionalnosti, mi bomo opisali samo najpomembnejše.

2.1.5.1 Vgnezdene poizvedbe

Vgnezdena poizvedba je poizvedba v sklop `WHERE` zunanje poizvedbe (slika 2.8). Rezultat vgnezdene poizvedbe uporabimo v pogoju. V notranji poizvedbi lahko uporabimo tudi atribute zunanje poizvedbe. Če ne definiramo kateri tabeli pripada ime stolpca v notranji poizvedbi, pripada tabeli v najbolj notranji poizvedbi (slika 2.9).

```
SELECT *
FROM T_KOPIJA
WHERE stevilka IN (SELECT stevilka
                   FROM A_KOPIJA)
```

Slika 2.8: Primer vgnezdene poizvedbe.

Za operator imamo veliko različnih možnosti. Če notranja poizvedba vedno vrne samo eno vrednost, lahko uporabimo tudi operatorje kot so enakost, primerjanje velikosti in podobno. V splošnem pa notranja poizvedba vrne tabelo, v teh primerih moramo uporabiti operatorje kot so `IN` (vrednost je v množici) ali `EXISTS` (obstaja). Paziti moramo tudi, da notranja poizvedba vrača enako število stolpcev kot jih bomo za primerjavo uporabili v zunanji poizvedbi (slika 2.9). Te funkcionalnosti ne podpirajo vse implementacije jezika SQL.

```
SELECT *
FROM T_KOPIJA
WHERE (stevilka, datum) IN (SELECT stevilka, datum
                            FROM A_KOPIJA
                            WHERE T_KOPIJA.vnasalec=vnasalec)
```

Slika 2.9: Primer poizvedbe z več stolpci v filtru.

2.1.5.2 Agregacijske funkcije

Agregacijske funkcije povzamejo podatke iz večih vrstic v eno. Povezovanje (ang. grouping) uporabimo za ustvarjanje podskupin vrstic pred agregacijo. Agregacijske funkcije so `COUNT` (štetje števila vrstic), `SUM` (vsota), `MAX` (maksimalna vrednost), `MIN` (minimalna vrednost) in `AVG` (povprečje). Te funkcije lahko uporabimo v stavku `SELECT` ali `HAVING`, ki deluje podobno kot `WHERE`, le da deluje nad združenimi podatki (slika 2.10).

```
SELECT stevilka, MAX(kopija) AS max_kopija
FROM T_KOPIJA
GROUP BY stevilka
HAVING MAX(kopija)>0
```

	stevilka	max_kopija
1	150000000004	1
2	150000000009	3
3	150000000015	2
4	150000000026	1
5	150000000029	2
6	150000000031	1
7	150000000032	1
8	150000000033	1

Slika 2.10: Primer združevanja podatkov in uporabe agregacijske funkcije ter rezultat poizvedbe.

2.1.5.3 Povezovanje

SQL pozna posebne ukaze za povezovanje večih tabel, kar se sicer da narediti tudi z naštevanjem tabel v sklopu `FROM` in s pogoji združevanja v sklopu `WHERE`. Lahko je pa povezovanje bolj pregledno s posebnimi ukazi. Poznamo več vrst povezovanja:

- Notranje (`INNER JOIN`): n-terica po kateri povezujemo mora obstajati v obeh relacijah.
- Levo odprto povezovanje (`LEFT OUTER JOIN`): n-terica po kateri povezujemo mora obstajati v prvi oz. levi relaciji (slika 2.11).
- Desno odprto povezovanje (`RIGHT OUTER JOIN`): n-terica po kateri povezujemo mora obstajati v drugi oz. desni relaciji.
- Polno odprto povezovanje (`FULL OUTER JOIN`): n-terica po kateri povezujemo mora obstajati vsaj v eni relaciji.
- Naravno povezovanje (`NATURAL`): če imata dve relaciji enaka imena atributov po katerih združujemo, lahko uporabimo besedo `NATURAL` in samo naštejemo ta imena atributov. Naravno povezovanje lahko uporabimo pri notranjem ali zunanjem povezovanju.
- Kartezijski produkt (`CROSS JOIN`): rezultat vsebuje vse možne kombinacije n-teric iz ene in druge relacije.

```
SELECT stevilka, tiskalnik
FROM T_KOPIJA LEFT OUTER JOIN uporabnik ON vnasalec=naziv
```

	stevilka	tiskalnik
1	150000000001	NULL
2	150000000002	NULL
3	150000000003	NULL
4	150000000004	NULL

Slika 2.11: Primer levo odprtega združevanja. Vrstice prve tabele se pojavijo v rezultatu tudi če ne obstaja ustrezna vrstica v drugi tabeli.

2.1.5.4 Pogledi

Pogled (ang. view) je tabela izpeljana iz drugih tabel. Druge tabele so lahko prave tabele ali že prej definirani pogledi. Pogled ni nujno fizično shranjen v podatkovni bazi, lahko je virtualna tabela. Pogled uporabimo, če pogosto povezujemo več tabel na isti način. Takšno poizvedbo lahko shranimo v pogled (slika 2.12) in izvajamo poizvedbe nad tem pogledom enako kot jih bi nad tabelo.

```
CREATE VIEW OSEBA_VIEW
AS
SELECT ose_emso AS emso, u_uporabnik AS sifra, u_podpisnik AS ime, 1 AS aktiven
FROM T_UPORABNIK INNER JOIN T_OSEBA ON u_uporabnik=ose_sif_sta
```

Slika 2.12: Primer ukaza za kreiranje novega pogleda.

Podatki v pogledu so vedno aktualni, usklajeni s podatki v tabelah iz katerih je izpeljan. Nad pogledamo lahko izvedemo tudi ukaz `UPDATE`, vendar je za to nekaj pogojev: pogled mora biti definiran nad eno tabelo, vsebovat mora vse atribute primarnega ključa in vse atribute, ki morajo vsebovati vrednost v tabeli iz katere je izpeljan, če ti atributi nimajo določene privzete vrednosti. Ostali pogledi v splošnem ne omogočajo ukaza `UPDATE`.

2.1.5.5 Prožilci

Prožilci (ang. triggers) omogočajo definiranje akcije, ki se naj zgodi ob določenem dogodku. Prožilec definiramo nad neko tabelo in določimo ob katerih dogodkih se naj zažene (slika 2.13). Možni dogodki so vstavljanje nove vrstice, posodabljanje vrstice oz. določenih stolpcev in brisanje vrstice. S prožilci lahko spremljamo spremembe podatkov v bazi in skrbimo za njihovo konsistenco.

```
CREATE TRIGGER trigger_kopije
ON A_KOPIJA
FOR UPDATE, INSERT
AS
INSERT INTO OnLine_log SELECT * FROM inserted
```

Slika 2.13: Primer kreiranje prožilca v sintaksi SQL Server. Tabela `inserted` je predhodno definirana tabela. Ima enako strukturo kot tabela nad katero se izvaja prožilec, vsebuje pa podatke, ki so se pravkar vstavili v tabelo ali spremenili.

2.1.5.6 Trditve

Standard SQL definira tudi trditve (`ASSERTION`), česar pa ne podpirajo vse implementacije. Trditev je še način definiranja omejitev v podatkovni bazi. Je dopolnitev preverjanju omejitev (`CHECK CONSTRAINT`) in prožilcem. Deluje tako, da definiramo neko trditev, ki mora v podatkovni bazi vedno veljati (slika 2.14). Akcija, ki bi podatkovno bazo postavila v stanje v nasprotju s trditvijo, ne bo uspela.

```
CREATE ASSERTION kopija_constraint
CHECK (NOT EXISTS (
            SELECT *
            FROM A_KOPIJA
            WHERE kopija>5
            )
        )
```

Slika 2.54: Primer kreiranje trditve.

2.1.5.7 Shranjene procedure in funkcije

Shranjene procedure in funkcije so definirane v podatkovni bazi, tako da se jih lahko uporablja v poizvedbah ali kako drugače, npr. preko ukaza `CALL` v interaktivnem vmesniku.

Uporabljamo jih lahko iz različnih aplikacij, ki uporabljajo isto podatkovno bazo. Uporabimo jih lahko tudi za bolj kompleksno preverjanje omejitev nad podatki, ki jih ne omogočajo prožilci ali trditve.

V procedurah in funkcijah lahko uporabljamo konstrukte kot jih poznamo v ostalih proceduralnih programskih jezikih; vejitve, zanke in klice podprogramov.

Procedura lahko kot rezultat vrne tabelo, samo eno vrednost ali ničesar. Funkcija vedno vrne vrednost ali tabelo.

2.2 XQuery

XQuery je specifikacija jezika za poizvedovanje in funkcionalno programiranje, ki poizveduje in transformira zbirke strukturiranih in nestrukturiranih podatkov, po navadi v obliki XML. Konkretne implementacije jezika lahko podpirajo tudi poizvedbe nad drugačnimi podatki (JSON, binarni, relacijske baze, ...). Jezik razvija delovna skupina pri konzorciju W3C [6].

2.2.1 Zmožnosti jezika

Jezik ima omogoča veliko operacij nad podatki XML, med drugim [8]:

- izbiranje podatkov glede na kriterije;
- filtriranje nepotrebnih podatkov;
- iskanje podatkov po dokumentu ali večih dokumentih;
- povezovanje podatkov iz več zbirk dokumentov;
- sortiranje, grupiranje in agregacija podatkov;
- transformiranje podatkov XML v drugačno obliko XML;
- aritmetične operacije nad števili in datumi;
- preoblikovanje nizov;

2.2.2 Sintaksa

Jezik ima dve sintaksi, ena je namenjana programiranju, druga pa je v obliki XML [6].

Človeku lažje razumljiva sintaksa uporablja izraze FLWOR. Poimenovana je po vrstnemu redu ključnih besed:

- `FOR` definira zaporedje vozlišč. V izrazu imamo vsaj en `FOR` ali let stavek, lahko pa tudi več. V njem definiramo spremenljivko in seznam vozlišč ali osnovnih vrednosti. Za vsak element v seznamu se spremenljivki nastavi vrednost s katero se izvede ostali del izraza FLWOR. Če imamo več stavkov `FOR`, smo s tem naredili tudi povezovanje. Povezovanje v tem primeru omejimo v sklopu `WHERE`.

- `LET` poveže vozlišče s spremenljivk podobno kot `FOR`, le da spremenljivka dobi neko vrednost. Sklop ni obvezen, če obstaja vsaj ena sklop `FOR`. Služi kot pripomoček pri programiranju. Spremenljivki dodelimo neko vrednost, ki jo lahko od tu naprej uporabljamo, da nam ni treba večkrat po poizvedbi pisat daljšega izraza (slika 2.1). Pomaga lahko pri hitrosti izvajanja, ker se izraz izračuna samo enkrat. Tudi na ta način lahko naredimo povezovanje večih podatkov (slika 2.15).

- `WHERE` filtrira vozlišča. Izgleda podobno kot v ostalih poizvedovalnih jezikih. Sklop ni obvezen.

- `Order by` sortira vozlišča.

- `Return` vrne vrednost poizvedbe za vsako vozlišče.

```
for $b in doc ("racuni.xml" )/racuni/racun
let $kupec := concat($b/nazivKupca, ' ', $b/naslovKupca), $pozicije := $b/pozicije
where $b/leto= 2016
order by $b/stevilkaRacuna
return
    <racun>
        {$b/datum}
        {$b/znesek}
        <kupec>{$kupec}</kupec>
        <pozicije>
            {$pozicije/pozicija/opis}
        </pozicije>
    </racun>
```

Slika 2.152.6: Primer izraza v obliki FLWOR.

Nekatere ključne besede lahko povežemo z jezikom SQL. Sklop `WHERE` tudi v tem jeziku predstavlja filter, `order by` in `group by` sta podobna. Razlika pa je v izbiri vira podatkov in predstavitvi. V jeziku SQL v sklopu `SELECT` izberemo podatke oz. stolpce, ki jih hočemo v rezultatu, v izrazu FLWOR to naredimo v sklopu return, ki je pa drugačne oblike. Tukaj ne naštevamo samo stolpcev, ampak lahko rezultat oblikujemo tudi kako drugače kot v plosko tabelo. Če poženemo poizvedbo iz slike 2.15, dobimo rezultat na sliki 2.16.

```
<racun>
  <datum>2.2.2016</datum>
  <znesek>14.50</znesek>
  <kupec>Mojca Šolska 3, 1000 Ljubljana</kupec>
  <pozicije>
    <opis>Torba</opis>
    <opis>Svinčnik</opis>
  </pozicije>
</racun>
```

Slika 2.76: Rezultat poizvedbe iz slike 2.15.

Vidimo, da rezultat ni navadna ploska tabela s stolpci in njihovimi vrednostmi, ampak nov dokument XML, ki je sicer lahko tudi seznam enostavnih vrednosti, lahko je pa hierarhičen kot v tem primeru.

Na sliki 2.15 imamo poizvedbo, ki črpa podatke iz dokumenta XML, ki že vsebuje hierarhične podatke. Če pa imamo več virov podatkov in hočemo iz njih narediti nov hierarhičen rezultat, moramo napisati poizvedbo kot je na sliki 2.16 [11]. V tem primeru dobimo hierarhični rezultat (slika 2.17).

```
for $b in doc("racuni.xml")/racuni/racun
let $r := $b/stevilka
return
<racun>
  {$r}
  <pozicije>
    {
      for $c in doc("cenik.xml")/artikli/artikel
      let $p := $b/pozicije,  $s := $c/sifra
      where $p/pozicija/sifra=$c/sifra
      return
        <pozicija>
          {$c/opis, $c/cena}
        </pozicija>
    }
  </pozicije>
</racun>
```

Slika 2.16: Poizvedba z vgnezdeno poizvedbo. V jeziku SQL bi se takšno povezovanje imenovalo levo odprto.

```
<racun>
  <stevilka>2</stevilka>
  <pozicije>
    <pozicija>
      <opis>Torba</opis>
      <cena>20.00</cena>
    </pozicija>
    <pozicija>
      <opis>Svinčnik</opis>
      <cena>1.00</cena>
    </pozicija>
  </pozicije>
</racun>
```

Slika 2.17: Rezultat poizvedbe iz slike 2.16.

2.2.2.1 Poizvedbe v obliki XML

Poizvedbo lahko zapišemo tudi v obliki XML (slika 2.18). Ta oblika ni primerna za programiranje, ker je človeku težko berljiva, je pa enostavna za avtomatsko procesiranje. Če imamo podatke v obliki XML, in hočemo dobiti rezultate poizvedbe v obliki XML, je včasih koristno, da je tudi sama poizvedba v enaki obliki [6].

```
<q:flwr>
  <q:forAssignment variable="$b">
    <q:step axis="slashslash">
      <q:function name="document">
        <q:constant datatype="charstring">racuni.xml</q:constant>
      </q:function>
      <q:identifier>racuni</q:identifier>
      <q:identifier>racun</q:identifier>
    </q:step>
  </q:forAssignment>
  <q:where>
    <q:function name="equals">
      <q:step axis="child">
        <q:variable>$b</q:variable>
        <q:identifier>leto</q:identifier>
      </q:step>
      <q:constant datatype="integer">2016</q:constant>
    </q:function>
  </q:where>
  <q:return>
    <q:elementConstructor>
      <q:tagName>
        <q:identifier>racun</q:identifier>
      </q:tagName>
      <q:elementConstructor>
        <q:tagName>
          <q:identifier>datum</q:identifier>
        </q:tagName>
        <q:step axis="CHILD">
          <q:variable>$b</q:variable>
          <q:identifier>datum</q:identifier>
        </q:step>
      </q:elementConstructor>
      <q:elementConstructor>
        <q:tagName>
          <q:identifier>znesek</q:identifier>
        </q:tagName>
        <q:step axis="CHILD">
          <q:variable>$b</q:variable>
          <q:identifier>znesek</q:identifier>
        </q:step>
      </q:elementConstructor>
    </q:elementConstructor>
  </q:return>
</q:flwr>
```

Slika 2.88: Poizvedba v obliki XML.

2.2.3 Podatkovni Model

- Podatkovni model podpira preslikovanje iz Infoset oz. PSVI v XQuery. Infoset je abstraktna predstavitev podatkov dokumenta XML. Lahko si ga predstavljamo v obliki drevesa. Sestavljajo ga podatki deklaracije XML (kodiranje znakov, verzija,…), podatki o korenu dokumenta, vozliščih in atributih. PSVI (Post-Schema-Validation Infoset) je razširitev Infoseta. Doda še podatke o podatkovnih tipih. Podatkovni model XQuery je osnovan na modelu PSVI, dodaja mu pa še nekaj podatkovnih tipov in možnost predstavitve tudi slabo formiranih dokumentov XML, ker rezultat poizvedbe v XQuery ni nujno pravilen dokument XML. Prav tako mora XQuery podpirati vmesne rezultate in vhodne podatke, ki se jih ne da predstaviti z Infosetom [6].

- Podatkovni model podpira zaporedja. V jeziku XQuery je vsak element zaporedje, ki je lahko sestavljeno iz nič, enega ali več elementov [6].

- Poizvedbe se izvedejo tudi če shema ni na voljo. Dokument XML ima lahko definirano shemo, ki opisuje podatkovne tipe in omejitve podatkov v njem. Če je shema definirana, jo XQuery uporabi za validacijo podatkov [6].

Primerek podatkovnega modela imenujemo vrednost v kontekstu podatkovnega modela. To so na primer razčlenjeni dokumenti XML, osnovne vrednosti, zaporedja vozlišč pomešanih z osnovnimi vrednostmi ali pa zaporedja atributov. Vsak primerek podatkovnega modela v XQuery je zaporedje. Element zaporedja je ali vozlišče ali osnovna vrednost. Osnovna vrednost je vrednost iz nabora vrednosti osnovnih tipov. Osnovni tip je primitiven enostaven tip ali pa tip izpeljan iz drugega osnovnega tipa [6].

2.3 LINQ

Namen jezikovno integriranega povpraševalnega jezika (ang. language integrated query) LINQ je enostavnejše povezovanje podatkov v podatkovni bazi ali iz katerega drugega vira z objekti v programu.

Operacija poizvedbe poteka v treh korakih [9]:

- Pridobitev vira podatkov: vir mora podpirati določen vmesnik, v primeru okolja .NET `IEnumerable<T>` ali vmesnik izpeljan iz njega.

- Poizvedba: določi katere podatke se bo pridobilo iz vira podatkov.

- Izvedba poizvedbe: lahko se zgodi kasneje, ko se podatki iz poizvedbe zares potrebujejo. Če želimo takojšno izvedbo, lahko pokličemo funkcijo, ki prešteje število zapisov v rezultatu, ali pa rezultat spremenimo v seznam oz. polje. V tem primeru se rezultati shranijo v začasni pomnilnik in ob naslednji klicih so že pripravljeni.

2.3.1 Lastnosti

Bistvo jezika LINQ lažje razumemo preko sedmih lastnosti, ki jih bomo opisali [1].

2.3.1.1 Integriranost

Kot že ime pove, je LINQ integriran v sam programski jezik. S tem pridobimo, da se pravilnost poizvedbe delno preverja že med prevajanjem programa. Obenem na enoten način pišemo poizvedbe nad vsemi viri podatkov (podatkovne baze, XML, spletne storitve, objekti v programu, itd.).

2.3.1.2 Deklarativnost

LINQ je, podobno kot SQL (slika 2.19), deklarativni jezik, kar pomeni, da z njim opišemo rezultat, ki ga hočemo, in ne postopka, kot pri proceduralnem programiranju. Na ta način je oblika programa preglednejša, krajša, ni potrebno poznati vseh podrobnosti izvedbe, za optimizacijo poizvedbe pa poskrbi prevajalnik.

```
var seznamRacunov = from r in ds.Racun
                    where r.DatumRacuna = DateTime.Now.Date
                    select r;
```

Slika 2.99: Primer poizvedbe LINQ.

2.3.1.3 Hierarhičnost

Rezultat poizvedbe SQL je vedno v obliki tabele, kvadratne mreže. Rezultat poizvedbe v LINQ-u je pa seznam objektov (slika 2.20), ki pa so lahko hierarhični, kar pomeni, da nek objekt v rezultatu lahko vsebuje seznam drugih objektov. Tako so relacije v podatkih prikazane tudi v rezultatu, kar pri poizvedbi SQL ni mogoče.

```
var davkiNaRacunu = from d in seznamRacunov
                    where d.StevilkaRacuna = referencniRacunStevilka
                    select d.GetRacunDavkiRows();
```

Slika 2.20: Primer poizvedbe LINQ, ki vrne seznam objektov v objektu.

2.3.1.4 Sestavljivost

Poizvedbe LINQ so sestavljive, kar pomeni, da jih lahko med seboj združujemo. Rezultat ene poizvedbe lahko uporabimo v drugi (slika 2.21). S tem večji problem razbijemo na manjše enostavnejše probleme. Koda je enostavnejša in lažja za vzdrževanje, obenem se pa tudi izvede hitreje, če rezultate manjših poizvedb večkrat uporabimo.

```
var seznamRacunov = from r in ds.Racun
                    where r.DatumRacuna = DateTime.Now.Date
                    select r;
var blagajne = from r in seznamRacunov
                         where r.OznakaPoslovnegaProstora = poslovniProstor
                         select r.OznakaBlagajne;
```

Slika 2.21: Poizvedba LINQ, ki uporabi rezultat prve poizvedbe.

2.3.1.5 Transformativnost

LINQ ne omogoča samo poizvedbe nad podatki, ampak enostavne transformacije iz ene oblike v drugo. Enostavno je tudi ustvariti nov tipe, npr.:

- Rezultate večih poizvedb lahko združimo v eno zaporedje novega tipa.
- Ustvarimo lahko novo zaporedje elementov, ki imajo samo nekatere lastnosti elementov vhodnega zaporedja.
- Ustvarimo lahko zaporedje v drugem formatu, npr. podatke iz tabele pridobljene z jezikom SQL spremenimo v obliko XML.

2.3.1.6 Združujoč

LINQ omogoča enotno metodo za poizvedovanje po različnih virov podatkov. Z njim lahko nadomestimo poizvedbe v podatkovno bazo s SQL-om, XQuery in ostale jezike za poizvedbe nad podatki v obliki XML, omogoča poizvedbe nad spletnimi storitvami ter tudi nad seznami v samem programu (slika 2.22).

```
var racuni1 = from r in GetRacuni()
              where r.Datum == DateTime.Now.Date
              select new { r.StevilkaRacuna, r.ZnesekRacuna };

var racuni2 = from r in ds.Racun
              where r.Datum == DateTime.Now.Date
              select new { r.StevilkaRacuna, r.ZnesekRacuna };

var racuni3 = from r in racXml.Descendants("Racun")
              where r.Attribute("Datum").Value == DateTime.Now.ToShortDateString()
              select new {
                  StevilkaRacuna = r.Attribute("StevilkaRacuna").Value,
                  ZnesekRacuna = r.Attribute("ZnesekRacuna").Value
              };
```

Slika 2.22: Poizvedbe nad podatki iz samega programa, podatkovne baze in datoteke XML. Poizvedbe niso čisto enake, so pa zelo podobne.

To ima dve prednosti:

- Podobna sintaksa omogoča hitrejši razvoj ko potrebujemo podatke iz različnih virov. Ni se potrebno učiti različnih drugih poizvedovalnih jezikov.
- Kodo programa je lažje vzdrževati, ker je podobna za vse vire podatkov.

2.3.1.7 Razširljivost

LINQ ni samo poizvedovalni jezik, je tudi tehnologija za definiranje ponudnikov s katerimi dostopamo do različnih virov podatkov. LINQ že vsebuje ponudnike za poizvedbe SQL, nad datotekami XML in objekti, lahko pa vsakdo doda še svojega. Ostali zanimivi viri podatkov, ki jih omogočajo ponudniki, ki jih ni naredil Microsoft so še Google, Amazon, MySQL, WebQueries, Excel, CRM, Nhibernate, TerraServer (slika 2.23) in mnogi drugi.

Narediti ponudnika za dostop do novega vira podatkov ni enostavno, je pa enostaven za uporabo ko je enkrat narejen.

```
var query = from place in terraPlaces
            where place.Name == "Ljubljana"
            select new { place.Name, place.State };
```

Slika 2.103: Primer poizvedbe nad TerraPlaces. O zgradbi podatkov ne rabimo vedeti praktično ničesar, pa že lahko začnemo s pisanjem poizvedbe.

Še nekaj prednosti, ki jih omogoča razširljivost:

- Ponudnike je enostavno pregledati in spremeniti. Večina kode je dostopna razvijalcem.
- Poizvedbe lahko optimiziramo, ker lahko vidimo, kako se bodo izvedle.
- Naredimo lahko svoj ponudnik, npr. če ponujamo neko spletno storitev.

2.3.2 Lastnosti jezika, ki omogočajo LINQ

Programski jezik mora podpirati določene značilnosti, ki omogočajo LINQ [10].

2.3.2.1 Izraz poizvedbe

Izraz poizvedbe ima podobno sintakso kot SQL (slika 2.24). Z njim določimo iz kod se naj vzamejo podatki in kako se naj preoblikujejo.

```
var racuniNaDatum = from r in ds.Racun
                    where r.DatumRacuna == DateTime.Now.Date
                    orderby r.StevilkaRacuna
                    select r.StevilkaRacuna;
```

Slika 2.114: Primer poizvedbe LINQ.

2.3.2.2 Implicitno določen tip spremenljivke

Namesto, da bi eksplicitno določili tip spremenljivke, jo lahko deklariramo implicitno, tip pa določi prevajalnik iz izraza, ki določa njeno vrednost (slika 2.25). Vrednost spremenljivke mora biti obvezno določena že ob deklaraciji. Kasneje to vrednost lahko spremenimo, mora pa vedno ostati tipa int.

```
var celoStevilo = 0;
```

(local variable) int celoStevilo

Slika 2.25: Spremenljivka nima eksplicitno določenega tipa, vidimo pa, da je prevajalnik določil tip int.

2.3.2.3 Inicializatorji objektov in seznamov

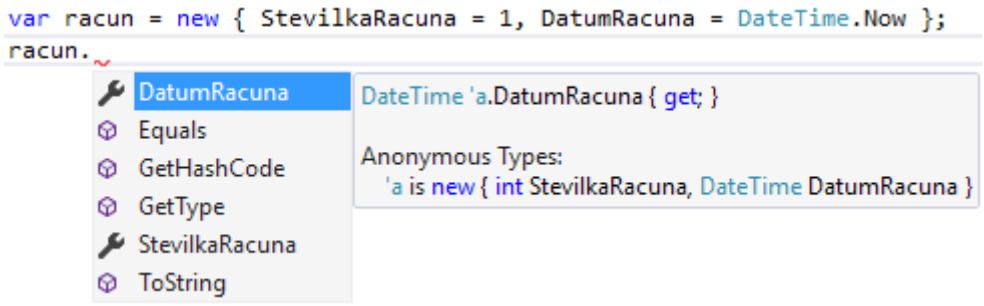
Omogočajo inicializacijo brez eksplicitnega klica konstruktorja objekta. Uporabljajo se v izrazu poizvedbe, kadar želimo projicirati vir podatkov v nov podatkovni tip (slika 2.26).

```
Racun rac = new Racun { StevilkaRacuna = "01", DatumRacuna = DateTime.Now };
```

Slika 2.26: Primer inicializacije objekta brez eksplicitnega klica konstruktorja.

2.3.2.4 Anonimni tipi

Z njimi ustvarimo nov tip, ki ga pozna samo prevajalnik, brez da bi ga posebej definirali (slika 2.27).



Slika 2.27: Spremenljivka racun nima eksplicitno določenega tipa. Nov tip je ustvaril prevajalnik, ta tip pa vsebuje lastnosti DatumRacuna in StevilkaRacuna, ki jih je prevajalnik avtomatsko kreiral.

2.3.2.5 Razširitvene metode

Povezane so z določenim tipom tako, da jih je možno klicati nad objekti tega tipa kot da so njegove lastne.

```
public static class RazsiritveneMetode
{
    public static bool Potrdi(this Racun racun)
    {
        return true;
    }
}

Racun rac = new Racun { StevilkaRacuna = "01", DatumRacuna = DateTime.Now };
rac.Potrdi();
```

Slika 2.128: Nad primerkom razreda `Racun` lahko kličemo metodo `Potrdi()`, čeprav je razred `Racun` ne vsebuje.

2.3.2.6 Izrazi lambda

Lambda izrazi so vrinjene anonimne funkcije, ki lahko ustvarijo delegate ali tipe dreves izražanja. Sestavljeni so iz parametra in izraza, ki izračuna vrednost izraza lambda (slika 2.29).

```
var racuniNaDatum = ds.Racun.Where(i => i.DatumRacuna == DateTime.Now.Date);
```

Slika 2.139: Primer izraza Lambda.

Z izrazi lambda je možno izraziti vse kar je možno izraziti tudi z izrazom poizvedbe, vsebujejo pa še dodatne metode, ki niso podprte v sintaksi izraza poizvedbe. Te dodatne metode lahko uporabimo tudi v poizvedbi tako, da v poizvedbo vključimo izraz lambda. Tudi sami lahko enostavno dodajamo nove razširitvene metode.

2.3.2.7 Samodejno implementirane lastnosti

Samodejno implementirane lastnosti omogočajo enostavno deklaracijo lastnosti, ki ji prevajalnik avtomatsko doda anonimno polje v katerem je shranjena njena vrednost. To anonimno polje je dostopno samo preko metod `get` in `set` (slika 2.29).

```
public int Count { get; set; }
```

Slika 2.29: Primer deklaracije lastnosti, ki naj bo samodejno implementirana.

2.4 Primerjava funkcionalnosti

Poizvedovalne jezike bi lahko primerjali po večih lastnostih: matematičnih, zmogljivosti oz. funkcionalnosti in po človeškem faktorju [7]. Mi bomo primerjali sintakso, ki vpliva na prijaznost do uporabnika, ter funkcionalnost.

2.4.1 Podatkovni model

Jezik SQL uporablja relacijski podatkovni model. Formalno je podatkovna baza zbirka relacij, relacija vsebuje množico n-teric, n-terice pa imajo določene atribute. To si lahko predstavljamo tudi kot tabelo, vrstico v tabeli in imena stolpcev. Podatki so strukturirani [4].

Vsak primerek podatkovnega modela v XQuery je zaporedje, ki je sestavljeno iz nič ali več elementov. Element je pa lahko en ali več dokumentov XML, osnovna vrednost, zaporedje vozlišč pomešanih z osnovnimi vrednostmi ali zaporedje atributov vozlišč. Podatki niso nujno strukturirani [6].

LINQ zahteva, da je struktura podatkov znana. Poizvedbe izvaja nad seznami objektov, ki morajo imeti znane lastnosti. Ni možno, da bi v istem seznamu objektov imeli objekte različnih tipov [1].

Vidimo, da ima XQuery najbolj fleksibilen podatkovni model. Podatki ne potrebujejo nujno znane strukture in lahko so urejeni hierarhično. V jeziku SQL podatki potrebujejo strukturo, in ne morejo biti hierarhični. LINQ zahteva strukturo, podatki pa so lahko tudi hierarhični.

2.4.2 Osnovne poizvedbe

V osnovni poizvedbi ne glede na poizvedovalni jezik nas po navadi zanimajo vsi podatki iz nekega vira (tabele, datoteke XML, ...). Dodajmo še urejenost po enem od atributov (slika 2.30).

SQL:
```
select * from racuni order by datum
```

XQuery:
```
for $b in doc("racuni.xml")/racuni/racun
order by $b/datum
return $b
```

LINQ – izraz poizvedbe:
```
from r in ds.Racun
orderby r.Datum
select r;
```

LINQ – izraz lambda:
```
ds.Racun.OrderBy(i => i.Datum);
```

Slika 2.30: Enostavna poizvedba v različnih poizvedovalnih jezikih.

Vidimo, da imajo vsi trije jeziki podobne ključne besede, le v XQuery imamo return namesto SELECT. Izraz lambda je drugačen, ker uporablja sintakso jezika gostitelja, v tem primeru C#, in razširitvene metode. Interno se izraz poizvedbe pretvori v izraz lambda.

Kot smo napisali že pri podatkovnem modelu, ima XQuery najbolj fleksibilnega, ker struktura podatkov ni nujno poznana. Če pri viru podatkov, ki se uporablja na sliki 2.18, ne bi obstajala relacija oz. tabela `racun`, bi nas v jeziku LINQ na to opozoril že prevajalnik, tudi v jeziku SQL bi do napake prišlo najkasneje pri izvajanju poizvedbe, poizvedba XQuery nam bi pa v tem primeru enostavno vrnila prazen rezultat, ne da bi vedeli ali podatkov res ni ali pa smo napačno zapisali poizvedbo. Tudi če ne bi obstajal atribut `Datum`, nas XQuery ne bi opozoril, le sortiral ne bi podatkov.

2.4.3 Povezovanje

Vsi trije jeziki podpirajo izbiro vira podatkov iz seznama virov in njihovo povezovanje. Pri jeziku SQL je vir vedno relacija ali več relacij v podatkovni bazi (slika 2.31) [4], pri XQuery in LINQ je vir poleg relacije lahko dokument XML, pri LINQ pa tudi seznam objektov. SQL in LINQ podpirata povezovanje s podobno sintakso (sliki 2.31 in 2.32), lahko pa v LINQ-u to naredimo tudi drugače in morda bolj berljivo (slika 2.33).

```
select StevilkaRacuna, Stopnja
from racuni left outer join davki on racuni.davek=davki.davek
```

Slika 2.31: Levo zunanje povezovanje v jeziku SQL.

```
var leftOuterJoin = from r in ds.Racun
                    join d in ds.RacunDavki on r equals d.RacunRow into rd
                    from d in rd.DefaultIfEmpty()
                    select new { r.StevilkaRacuna, d.Stopnja };
```

Slika 2.32: Levo zunanje povezovanje v jeziku LINQ z operatorjem JOIN.

```
var leftOuterJoin2 = from r in ds.Racun
                     from d in ds.RacunDavki.Where(i => i.RacunRow == r).DefaultIfEmpty()
                     select new { r.StevilkaRacuna, d.Stopnja };
```

Slika 2.33: Levo zunanje povezovanje v jeziku LINQ z uporabo izraza lambda namesto operatorja `JOIN`. Če odstranimo klic metode `DefaultIfEmpty`, dobimo notranje povezovanje.

Če je v jezikih SQL in LINQ notranje in zunanje povezovanje dokaj podobno v sintaksi, je v jeziku XQuery to malo drugače. Obstaja več načinov združevanja [11]. Notranje povezovanje je možno narediti kot je prikazano na sliki 2.34. Enostavno naštejemo več `FOR` stavkov in določimo po katerem atributu želimo združiti vire podatkov.

```
for $b in doc("racuni.xml")/racuni/racun
for $c in doc("cenik.xml")/artikli/artikel[sifra=$b/pozicije/pozicija/sifra]
return
<racun>
  {$b/stevilka, $c/sifra, $c/cena}
</racun>
```

Slika 2.34: Notranje povezovanje v jeziku XQuery. Rezultat poizvedbe vidimo na sliki 2.23.

```
<racun>
  <stevilka>2</stevilka>
  <sifra>1</sifra>
  <cena>20.00</cena>
</racun>
<racun>
  <stevilka>2</stevilka>
  <sifra>2</sifra>
  <cena>1.00</cena>
</racun>
```

Slika 2.145: Rezultat poizvedbe iz slike 2.34.

Zunanje povezovanje pa je potrebno narediti drugače. V bistvu ne gre za povezovanje, ki ga bi posebej podpiral sam jezik, ampak vgnezdeno poizvedbo v sklopu return glavne poizvedbe (slika 2.36).

```
for $b in doc("racuni.xml")/racuni/racun
return
<racun>
 {$b/stevilka}
    {
      for $c in doc("cenik.xml")/artikli/artikel
      where $b/pozicije/pozicija/sifra=$c/sifra
      return
        <artikel>
          {$c/sifra}
          {$c/cena}
        </artikel>
    }
</racun>
```

Slika 2.36: Zunanje povezovanje v jeziku XQuery.

```
<racun>
  <stevilka>1</stevilka>
</racun>
<racun>
  <stevilka>2</stevilka>
  <artikel>
    <sifra>1</sifra>
    <cena>20.00</cena>
  </artikel>
  <artikel>
    <sifra>2</sifra>
    <cena>1.00</cena>
  </artikel>
</racun>
<racun>
  <stevilka>3</stevilka>
</racun>
```

Slika 2.37: rezultat poizvedbe iz slike 2.36.

Če pogledamo sliki 2.35 in 2.37 lahko opazimo še eno stvar: rezultat iz slike 2.35 si lahko predstavljamo kot tabelo, ker imamo več vozlišč z imenom racun, in 3 atribute, ki jih bi lahko pretvorili v imena stolpcev. Zunanje povezovanje nam pa nujno vrne neko drevesno strukturo in ne tabele. Če hočemo za rezultat dobiti tabelo, kar je v veliko primerih smiselno, je potrebno celoten postopek zunanjega združevanja narediti še malo drugače. Najprej je potrebno poiskati zapise, ki nimajo pripadajočih podatkov v zapisih s katerimi jih združujemo. Tem zapisom vpišemo neko vrednost, ki nam bo predstavljala prazno vrednost. Te zapise pa združimo še z zapisi, ki imajo tudi pripadajoče zapise, kot je to prikazano na sliki 2.38. V tem primeru dobimo podatke v obliki, ki omogoča enostavno pretvorbo v tabelo (slika 2.39).

```
let $prazno :=
(for $b in doc("racuni.xml")/racuni/racun
 where empty(doc("cenik.xml")/artikli/artikel[sifra=$b/pozicije/pozicija/sifra])
 return
   <racun>
     {$b/stevilka}
     <sifra></sifra>
     <cena></cena>
   </racun>
)

let $polno :=
(for $b in doc("racuni.xml")/racuni/racun
 for $c in doc("cenik.xml")/artikli/artikel[sifra=$b/pozicije/pozicija/sifra]
 return
   <racun>
    {$b/stevilka}
    {$c/sifra}
    {$c/cena}
  </racun>
)
return $prazno|$polno
```

Slika 2.38: Primer zunanjega združevanja v jeziku XQuery, s katerim dobimo rezultat, ki ga lahko enostavno pretvorimo v tabelo.

```
<racun>
  <stevilka>1</stevilka>
  <sifra/>
  <cena/>
</racun>
<racun>
  <stevilka>3</stevilka>
  <sifra/>
  <cena/>
</racun>
<racun>
  <stevilka>2</stevilka>
  <sifra>1</sifra>
  <cena>20.00</cena>
</racun>
<racun>
  <stevilka>2</stevilka>
  <sifra>2</sifra>
  <cena>1.00</cena>
</racun>
```

Slika 2.39: Rezultat poizvedbe iz slike 2.38.

V vseh treh jezikih je povezovanje torej mogoče narediti, najbolje pa je podprto v jeziku SQL, ker ima že vgrajene operatorje kot so `INNER JOIN`, `LEFT OUTER JOIN`, `RIGHT`

OUTER JOIN, itd [2]. V jeziku LINQ in XQuery desno odprto povezovanje niti ni posebej podprto, ampak ga je potrebno narediti drugače. V jeziku LINQ bi bilo potrebno obrniti poizvedbo, da bi lahko naredili levo odprto povezovanje, v XQuery pa bi to lahko naredili po postopku, ki je prikazan na sliki 2.38.

2.4.4 Spreminjanje podatkov

V jeziku SQL [4] in XQuery [12] je možno spreminjati podatke s stavki INSERT, UPDATE in DELETE. V jeziku LINQ podatke spreminjamo tako, da jih s poizvedbo pridobimo, po želji spremenimo z metodami jezika gostitelja, nazaj pa shranimo z ukazom, ki je odvisen od tega kater vir podatkov smo uporabili. Če je to podatkovna baza, bo ta ukaz sam poskrbel, da se bodo poklicali ustrezni ukazi INSERT, UPDATE in DELETE [1].

2.4.5 Strukturiranje rezultata

Izbiro določenih elementov, ki jih hočemo v rezultatu poizvedbe, imenujemo projekcija. Vsi poizvedovalni jeziki jo podpirajo. V jeziku SQL in LINQ to naredimo z izrazom SELECT, v XQuery pa z return. Poizvedba SQL bo vedno vrnila plosko tabelo vrednosti. XQuery lahko poleg seznama vrednosti vrne tudi strukturo XML, ki lahko vsebuje tudi hierarhične podatke, v različnih oblikah, kot lahko vidimo na slikah 2.37 in 2.39. LINQ vedno vrača seznam objektov oz. primerkov nekega razreda, ki so tudi lahko hierarhično povezani. Vsi jeziki podpirajo tudi sortiranje rezultatov.

Glede te lastnosti sta proti jeziku SQL v prednosti jezika XQuery in LINQ, ker zmoreta vse kar zmore SQL, in več. Med XQuery in LINQ je sicer velika razlika v predstavitvi podatkov, težko je pa reči kater je zmogljivejši, to je odvisno od namena pridobivanja podatkov. Če so razredi nad katerimi delamo poizvedbe v jeziku LINQ tako definirani, jih lahko enostavno pretvorimo v obliko XML in s tem dobimo enako strukturo kot bi jo z poizvedbo v XQuery. Vendar v tem primeru lahko tudi rezultat v obliki XML, ki bi ga lahko pridobili s poizvedbo v XQuery, enostavno pretvorimo v primerke razreda, ki jih potem lahko uporabimo v programu enako, kot bi jih če bi jih pridobili z poizvedbo LINQ.

2.4.6 Agregacija

Vsi jeziki podpirajo grupiranje in agregacijske funkcije kot so minimum, maksimum, povprečje, vsota, štetje zapisov, itd.

2.4.7 Kompleksne poizvedbe

Podporo kompleksnim poizvedbam bomo preverili z naslednjimi lastnostmi [7]:

2.4.7.1 Sposobnost sledenja hierarhičnem poljem gnezdenih dokumentov

SQL za to potrebuje vgnezdeno poizvedbo za vsak nivo hierarhičnih podatkov [2]. V XQuery je to bolj enostavno, samo vpišemo pot do vrednosti ali atributa, ki nas zanima. Tudi v jeziku LINQ je to enostavno. Razred za katerega nas zanimajo podatki nižje ali višje v hierarhiji vsebuje funkcije ali lastnosti, ki nas tam pripeljejo.

2.4.7.2 Gnezdenje

Gnezdenje imenujemo možnost uporabe podpoizvedbe znotraj glavne poizvedbe [7]. To podpirajo vsi poizvedovalni jeziki. V jezikih SQL in LINQ lahko v izrazu `SELECT` namesto imena atributa zapišemo tudi novo poizvedbo. V XQuery podobno lahko v sklop `return` vstavimo novo poizvedbo (slika 2.36).

2.4.7.3 Podatkovni tipi

Vsi trije jeziki so močno tipizirani, kar pomeni, da vsaka funkcija in vsak operator pričakuje, da bo operand določenega tipa [1][4][6].

2.4.7.4 Razširljivost

Vsi trije jeziki podpirajo veliko različnih podatkovnih tipov. Pri vseh je pa tudi možno definirati nove, ali pa določiti razne omejitve za posamezen tip ali atribut, tako da glede razširljivosti pri podatkovnih tipih ni veliko razlike, čeprav so razlike na kakšen način to naredimo.

V vseh treh jezikih lahko tudi definiramo nove funkcije oz. procedure. V jeziku LINQ je ta funkcionalnost vgrajena v jezik gostitelja.

2.4.8 Operatorji

2.4.8.1 Vmesni operatorji

Vsi trije jeziki podpirajo vmesne operatorje. Za nekatere podatkovne tipe obstajajo še izrazi z besedo, na primer `concat(niz1, niz2)` v XQuery, ki spoji `niz1` in `niz2`, ali pa `equals` v LINQ, ki primerja vrednost prvega objekta z drugim.

2.4.8.2 Stavki nadzora poteka

Stavki v programu se normalno izvajajo od vrha proti dnu, v vrstnem redu kot so napisani. Stavki, ki spreminjajo ta potek, se imenujejo stavki nadzora poteka [7]. Vsi trije jeziki podpirajo stavke kot so `if-then-else` ter razne zanke, pa tudi izjave `try-catch`.

2.4.8.3 Operatorji za povezovanje

To so operatorji kot so unija (`UNION`), presek (`INTERSECTION`), razlika in kartezijski produkt. Vsi našteti operatorji so podprti v jeziku SQL [2]. V jeziku LINQ obstajajo razširitvene funkcije za unijo, presek in razliko, kartezijski produkt je pa potrebno narediti drugače, npr. z dvema `FROM` sklopoma [1]. Podobno je tudi v XQuery, ki pozna ukaze za unijo (|), presek (`intersect`) in razliko (`except`), pri tem je zanimivo, da za unijo uporablja znak, za ostala dva operatorja pa besedo. Tudi v XQuery je potrebno kartezijski produkt narediti npr. z dvema `FOR` sklopoma [11].

2.4.9 Podatki meta o poizvedbi

2.4.9.1 Transakcije

V jeziku SQL so transakcije dobro podprte z različnimi ukazi [2]. Sam XQuery nima popolne podpore za transakcije, zahteva samo, da je po končanem spreminjanju podatkov podatkovni model veljaven [12]. LINQ podpira transakcije preko jezika gostitelja, mora pa imeti podporo za to tudi vir podatkov [1].

2.4.9.2 Načrt poizvedbe

Praktično vse implementacije jezika SQL podpirajo ukaz za prikaz načrta izvedbe poizvedbe (`explain`). XQuery v specifikaciji [11] takšnega ukaza ne zahteva. Tudi LINQ takšnega ukaza ne pozna.

2.4.9.3 Indeksiranje

Indeksiranje uporabljajo podatkovne baze, da se izognejo preiskovanju vseh podatkov pri izvedbi poizvedbe. SQL ima obsežno podporo za indeksiranje [2]. LINQ lahko indeksiranje uporablja, če uporablja vir podatkov, ki ga podpira, nima pa posebnih ukazov za to. Posamezne implementacije XQuery tudi lahko indeksirajo podatke, ko se jim to zdi smiselno, ni pa zahtevano v specifikaciji [11].

2.4.10 Primerjava splošnih lastnosti

Poizvedovalne jezike smo primerjali še po petih splošnih lastnostih [5]:

- Ekspresivnost: pove kako močne poizvedbe lahko zapišemo v poizvedovalnem jeziku. Jezik naj bi podpiral najmanj relacijsko algebro.
- Zaprtost: ta lastnost zahteva, da je rezultat poizvedbe tudi primerek podatkovnega modela. To pomeni, da lahko z istim jezikom nadaljujemo z poizvedbami nad rezultatom prejšnje poizvedbe.
- Zadostnost: jezik je zadosten, če uporablja vse koncepte podatkovnega modela. Ta lastnost dopolnjuje zaprtost. Zaprtost zahteva, da je rezultat primerek podatkovnega modela, zadostnost pa, da je ves podatkovni model izkoriščen.
- Ortogonalnost: jezik je ortogonalen, če vse operacije v njem lahko uporabljamo neodvisno od konteksta.
- Varnost: jezik je varen, če sintaktično pravilna poizvedba vrne končen rezultat.

2.4.10.1 Ekspresivnost

Ekspresivnost posameznih jezikov smo opisali že pri opisu in primerjavi funkcionalnosti jezikov. Vsak jezik ima določene prednosti in slabosti, tudi namenjeni so za različne vire podatkov. Funkcionalnosti, ki jih nek jezik glede na specifikacijo ne podpira, so lahko v posameznih implementacijah rešene tudi z dodatnimi funkcijami. Lahko bi rekli, da so si glede ekspresivnosti jeziki dokaj podobni, SQL je v prednosti pri relacijskih podatkih, ostala dva pa sta v prednosti, če hočemo v rezultatu podatke prikazati hierarhično, ali jih pridobiti tudi še iz drugih virov, in ne samo iz podatkovne baze.

2.4.10.2 Zaprtost

Ta lastnost velja za vse 3 jezike. V jeziku SQL za rezultat dobimo tabelo, nad katero lahko izvedemo poizvedbo. Tudi LINQ za rezultat vrne seznam objektov, nad katerimi lahko naredimo novo poizvedbo. XQuery za rezultat vrne strukturo XML, nad katero tudi lahko naredimo novo poizvedbo.

2.4.10.3 Zadostnost

Jezik SQL izkorišča vse lastnosti svojega podatkovnega modela, saj je osnovan na relacijski algebri, ki predstavlja osnovo za operacije nad relacijskim modelom [4]. Tudi XQuery je zadosten, ker je namenjen izvajanju poizvedb nad zapisi v obliki XML, pri čemur omogoča dostop do vseh elementov, ki jih XML vsebuje, kot so npr. shema, imenski prostori, vozlišča

in atributi [6]. Zadostnost jezika LINQ je odvisna od implementacije posameznega podatkovnega vira. LINQ v povezavi s podatkovno bazo SQL ne podpira vseh možnosti baze. Sicer zmore prebrati na primer tip nekega atributa in osnovne omejitve (tip, dolžina niza in podobno), ne pa vseh. Pri zapisovanju nazaj v podatkovno bazo lahko pride do napake zaradi omejitve na bazi, ki jo LINQ ni mogel samodejno upoštevati [1].

2.4.10.4 Ortogonalnost

SQL je ortogonalen jezik. Če na primer izraz vrača število ali niz, ga lahko uporabimo kjerkoli lahko uporabimo število ali niz [2]. Enako velja za LINQ in XQuery.

2.4.11 Ostale lastnosti

Vsak jezik ima še veliko lastnosti specifičnih zanj. Tako na primer specifikacija SQL vsebuje tudi prožilce, poglede in nadzor nad dostopom. Ostala dva tega ne podpirata že v samem jeziku, ampak bi bilo to potrebno narediti z dodatnimi funkcijami.

Glavna prednost jezika LINQ je njegova integracija v jezik gostitelja. To omogoča, da z enim jezikom lahko naredimo celoten program, pa tudi uporabo različnih knjižnic, spletnih servisov in ostalih virov, ki jih lahko uporabimo v poizvedbah.

Še ena razlika med SQL in ostalima dvema jezika je v vrstnem redu stavkov v poizvedbi. SQL se začne s projekcijo, oz. stavkom `SELECT`, ostala dva pa z virom podatkov, torej `FROM` oz. `FOR`. Na začetku razvoja jezika SQL še niso pomislili na to, da nam bo leta kasneje integrirano razvojno okolje (angl. integrated development environment, IDE) znalo predlagati besede, ki jih bi morda želeli napisati v poizvedbo, glede na to kje smo znotraj poizvedbe. Če začnemo z naštevanjem virov podatkov, nam v naslednjih stavkih glede na vir IDE lahko predlaga imena atributov. Pri pisanju poizvedbe v jeziku SQL to ni mogoče, dokler ne napišemo sklopa `FROM`.

Poglavje 3 Dodelave sistema Algator

3.1 Algator

Algator je sistem za izvajanje algoritmov na podanih testnih podatkih ter analizo rezultatov izvajanja. Opisali bomo dele sistema, ki so povezani z analizo podatkov, povzeto iz njegove dokumentacije [3].

Algoritmi, testni podatki in rezultati izvajanja so definirani znotraj projekta. Sistem lahko vsebuje več projektov.

Projekt sistema Algator je definiran z:

- definicijo problema, testnih množic in vhodnih podatkov,
- opisom formata vhodnih in izhodnih podatkov,
- definicijo vseh javanskih razredov potrebnih za izvajanje algoritmov projekta.

Sistem avtomatsko izvaja v projektu definirane algoritme nad testnimi podatki in shrani rezultate izvajanja v obliki tabele. Nad rezultati se lahko izvajajo poizvedbe v obliki objekta JSON [3].

3.1.1 Analiza rezultatov

Format datoteke z rezultati je opisan v posebni datoteki projekta z imenom Result.atr. Datoteka je v obliki JSON. Vsebuje polja:

- `ParameterOrder`: vrstni red testnih parametrov.
- `IndicatorOrder`: vrstni red parametrov rezultata.
- `Parameters`: seznam parametrov rezultata s polji:
  - `Name`: kratko ime.
  - `Description`: daljši opis.
  - `Type`: tip parametra (celo število, niz, decimalno število, ...).

- `Indicators`: seznam parametrov rezultata z enakimi polji kot pri `Parameters`.

`ParameterOrder` in `IndicatorOrder` določita vrstni red parametrov v datoteki CSV. `Parameters` vsebujejo lastnosti testa (število elementov, ime grupe,…), uporabljajo se pri združevanju rezultatov različnih algoritmov. `Indicators` opisujejo rezultate izvedbe algoritma nad danim testom.

Datoteka z rezultati je v formatu CSV. Vsebuje naslednje podatke:

- ime algoritma;
- ime testne množice;
- enolična oznaka testa v testni množici;
- status;
- vhodne in izhodne parametra, v vrstnem redu kot je definiran v datoteki z opisom rezultata;

Če vrednost nekega parametra ni znana, vsebuje prazen niz. Če je pri izvajanju testa prišlo do napake, rezultat vsebuje prve 4 podatke (algoritem, testna množica, test, status), nato pa opis napake [3].

Program za izvajanje poizvedb se imenuje algator.Analyse. Program združi več datotek z rezultati v en seznam podatkov.

Poizvedbe so zapisane v formatu JSON (slika 3.1).

```
{
"Description" = "Demo query",
"Algorithms" = ["QuickSort AS QS", "BubbleSort AS BS"],
"TestSets" = ["TestSet1 AS TS1", "TestSet2 AS TS2"],
"Parameters" = ["TestSet", "Test AS T", "N"],
"Indicators" = ["TMin", "TMax"],
"GroupBy" = ["N; Tmax:SUM; MIN"],
"Filter" = ["N > 100", "N < 1000"],
"SortBy" = ["N:-"],
"Count" = ["1"]
}
```

Slika 3.1: Primer poizvedbe v obliki JSON.

Poizvedba vrne podatke v obliki tabele. Vsebuje elemente:

- Algoritmi znotraj projekta, ki jih hočemo analizirati: polje `algorithms`.
- Testne množice nad katerimi se bo izvedla poizvedba: `Testsets`.
- Parametre: `Parameters`.
- Indikatorje: `Indicators`.
- `Filter`.
- Grupiranje: `GroupBy`.
- Sortiranje: `SortBy`.
- Štetje: `Count`.

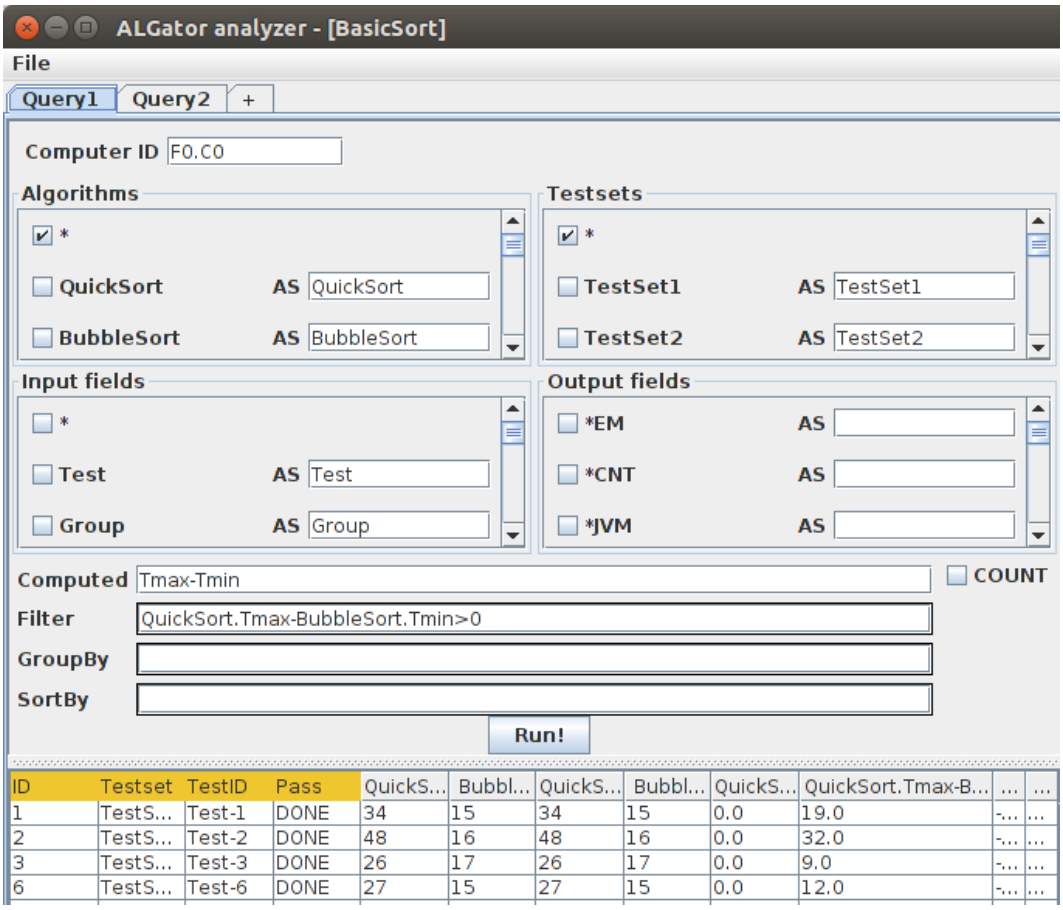
Algoritmi, testne množice, vhodni in izhodni parametri določajo obseg poizvedbe. Filter in grupiranje določajo število vrstic rezultata. Filter iz rezultata izloči vse vrstice, ki ne zadoščajo pogoju filtra. Filter lahko vsebuje parameter, npr. `Tavg < $1 @(10,100,5)`. Takšen filter se bo pognal večkrat, z vrednostmi parametra od 10 do 100, s korakom 5. Parameter v filtru je smiseln samo, če je vklopljeno številčenje.

Filter je tabela pogojev, v rezultatu pa ostanejo samo vrstice, ki ustrezajo vsem pogojem filtra. Operatorji filtra so `<`, `<=`, `>`, `>=`, `==`, `!=` za številske parametre in `==`, `!=` za znakovne.

Pri združevanju se vse vrstice z enako vrednostjo polja, po katerem združujemo, združijo v eno. Združevati je možno samo po parametrih testa.

3.1.2 Uporabniški vmesnik

Za izvajanje poizvedb lahko uporabimo uporabniški vmesnik (slika 3.2).



Slika 3.2: Uporabniški vmesnik za analizo rezultatov meritev.

Z uporabniškim vmesnikom izberemo algoritme, testne množice, vhodne in izhodne podatke. Nastavimo lahko še filter, grupiranje in sortiranje. Ob kliku na gumb Run se iz podatkov uporabniškega vmesnika sestavi poizvedba v formatu JSON in izvede. Rezultat se prikaže v tabeli.

3.2 Opis dodelav

V okviru diplomske naloge je bilo v sistem Algator dodanih nekaj funkcionalnosti in optimizacij.

3.2.1 Jezik A-SQL za poizvedbe

Dodana je bila možnost izvajanja poizvedb v jeziku A-SQL, ki je definiran znotraj sistema Algator. Podoben je jeziku SQL, ne vsebuje pa vseh njegovih elementov. Programu Analyse je dodana možnost parametra `-qf ASQL|JSON`, ki pove, v katerem formatu bo poizvedba. Privzet format ostaja JSON.

Za to funkcionalnost je bil v program Analyse dodan tip objekta `ASqlObject` s funkcijo `getJSONObject`, ki poizvedbo v formatu A-SQL prevede v format JSON. Od tu naprej se poizvedba izvede enako za oba formata. Dodana je tudi funkcija `initFromJSON`, ki sestavi objekt A-SQL iz poizvedbe v formatu JSON. S tem je omogočeno pretvarjanje iz ene oblike poizvedbe v drugo.

3.2.1.1 Sintaksa

Poizvedba v formatu A-SQL nima posebnih oznak za različne parametre poizvedbe kot v formatu JSON. Testne množice zapišemo v sklop `FROM`, ker so v testnih množicah definirani podatki. Algoritme zapišemo v sklop `WHERE`, obvezno v oklepaja, več jih naštejemo z operatorjem `OR`. Za definiranimi algoritmi obvezno sledi nova vrstica in `AND`, nato pa je zapisan še ostali del filtra, kot v polju `Filter` formata JSON. V sklopu `SELECT` lahko naštejemo tako vhodne kot izhodne parametre poizvedbe. Program bo parameter avtomatsko razvrstil med vhodne, če je oznaka parametra definirana v projektu kot vhodni parameter, sicer bo parameter izhodni. V `SELECT` lahko zapišemo tudi izračunana polja. Potem lahko sledita še sklopa `GROUPBY` IN `ORDERBY`, ki delujeta enako kot v formatu JSON.

Primer poizvedba v formatu A-SQL:

```
FROM TestSet1, TestSet2
WHERE (algorithm=JJava7 OR algorithm=CJava7)
AND Tmin > 100
AND ComputerID=F1.C1
SELECT N, Tmin AS Tmin, Tmax AS Tmax
```

Enakovredna poizvedba v formatu JSON:

```
{"Query":{
  "Name":"?",
  "Algorithms":[
      "JJava7 AS JJava7",
      "CJava7 AS CJava7"
  ],
  "TestSets": [
      "TestSet1 AS TestSet1",
      "TestSet2 AS TestSet2"
  ],
  "Parameters": [N],
  "Indicators": [
      "Tmin AS Tmin",
      "Tmax AS Tmax"
  ],
  "GroupBy": [""],
  "Filter": ["Tmin>100"],
  "SortBy": [""],
  "Count": "0",
  "ComputerID": "F1.C1"
}}
```

3.2.1.2 Sestavljivost

Poizvedbe v formatu A-SQL je možno sestavljati (slika 3.3) podobno kot to lahko delamo v jeziku LINQ.

```
queryF1C1 = FROM TestSet0
  WHERE (algorithm=*)
  AND ComputerID=F1.C1
  SELECT N, Tmax AS A1;

queryF1C2 = FROM TestSet0
  WHERE (algorithm=*)
  AND ComputerID=F1.C1
  SELECT N, Tmin AS A2;

FROM queryF1C1, queryF1C2
  WHERE (algorithm=JJava7 OR algorithm=CJava7)
  SELECT N, A1/A2 AS Q
```

Slika 3.3: Primer sestavljene poizvedbe.

Če hočemo narediti poizvedbo, ki črpa podatke iz drugih poizvedb, moramo prejšnjim poizvedbam določiti spremenljivke. V končni poizvedbi uporabimo te spremenljivke v sklopu `FROM`, v katerega tudi pri navadnih poizvedbah vpišemo vir podatkov oz. imena testnih množice. Rezultat sestavljene poizvedbe je podoben rezultatu navadne poizvedbe (slika 3.4).

ID	Testset	TestID	Pass	N	JJava7.Q	CJava7.Q
1	TestSet0	Test-1	DONE	100	7	1
2	TestSet0	Test-2	DONE	200	46	1
3	TestSet0	Test-3	DONE	300	3	1
4	TestSet0	Test-4	DONE	400	2	1
5	TestSet0	Test-5	DONE	500	6	1
6	TestSet0	Test-6	DONE	600	23	1
7	TestSet0	Test-7	DONE	700	6	1
8	TestSet0	Test-8	DONE	800	31	1
9	TestSet0	Test-9	DONE	900	22	1
10	TestSet0	Test-10	DONE	1000	13	1

Slika 3.4: Rezultat poizvedbe iz slike 3.3.

Poimenovane poizvedbe in testne množice je mogoče uporabiti tudi naenkrat v sestavljeni poizvedbi, vendar je rezultat takšne poizvedbe v splošnem unija poizvedbe in testne množice.

```
queryF1C1 = FROM TestSet0
  WHERE (algorithm=*)
  AND ComputerID=F1.C1
  SELECT N, Tmin;

FROM queryF1C1, TestSet1
  WHERE (algorithm=JJava7 OR algorithm=CJava7)
  AND ComputerID=F1.C1
  SELECT N, Tmin, Tmax
```

ID	Testset	TestID	Pass	N	JJava7.Tmin	CJava7.Tmin	JJava7.Tmax	CJava7.Tmax
1	TestSet0	Test-1	DONE	100	42	17	?	?
2	TestSet0	Test-2	DONE	200	43	38	?	?
3	TestSet0	Test-3	DONE	300	137	50	?	?
4	TestSet0	Test-4	DONE	400	209	73	?	?
5	TestSet0	Test-5	DONE	500	405	85	?	?
6	TestSet0	Test-6	DONE	600	266	130	?	?
7	TestSet0	Test-7	DONE	700	180	132	?	?
8	TestSet0	Test-8	DONE	800	211	154	?	?
9	TestSet0	Test-9	DONE	900	282	176	?	?
10	TestSet0	Test-10	DONE	1000	324	188	?	?
11	TestSet1	Test-1	DONE	100	41	6	365	11
12	TestSet1	Test-2	DONE	100	39	6	362	9
13	TestSet1	Test-3	DONE	100	42	5	365	8
14	TestSet1	Test-4	DONE	100	42	5	364	8
15	TestSet1	Test-5	DONE	100	40	5	364	8

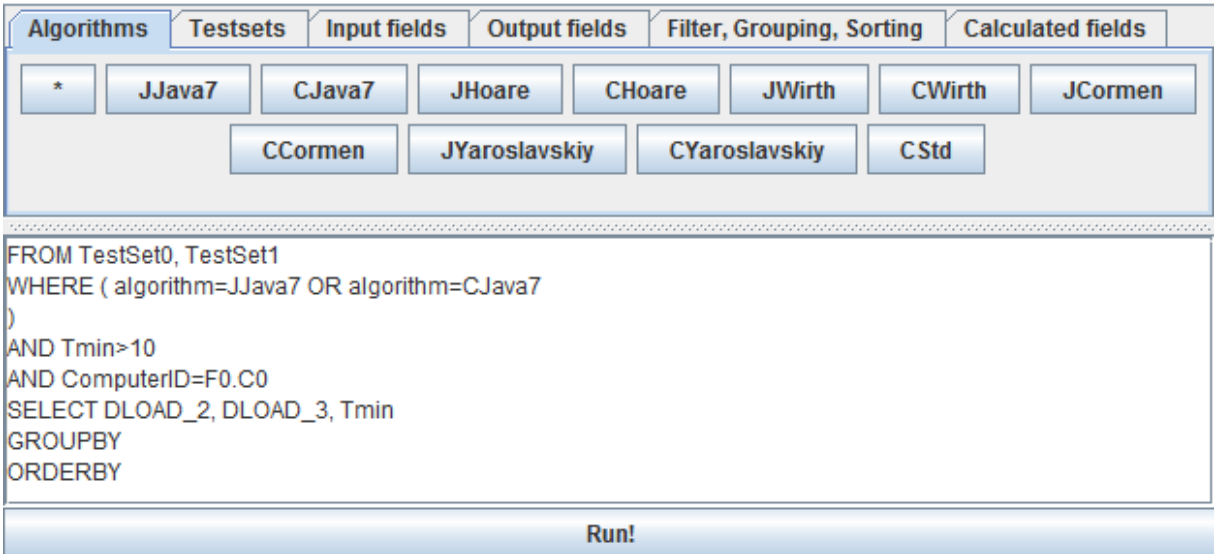
Slika 3.5: Poizvedba sestavljena iz poimenovane poizvedbe in testne množice ter rezultat.

Na sliki 3.5 vidimo, da poizvedba `queryF1C1`, ki zajema podatke iz `TestSet0`, vsebuje indikator `Tmin`, prav tako ta podatek vsebuje `TestSet1`, zato je prisoten v vseh vrsticah. Tmax pa ni prisoten v `queryF1C1`, zato imamo pri `TestSet0` v teh stolpcih prazne vrednosti.

Sestavljene poizvedbe delujejo tako, da se poizvedbe izvajajo ena za drugo, vsaka naslednja pa dobi imena in rezultate vseh prejšnjih. Te rezultate obravnava podobno kot testne množice, ki so sicer vir podatkov.

3.2.1.3 Grafični uporabniški vmesnik

Na uporabniški vmesnik sistema Algator je dodan urejevalnik poizvedb v formatu A-SQL (slika 3.6).



Slika 3.6: Uporabniški vmesnik za urejanje poizvedbe v formatu A-SQL.

Okno je razdeljeno na dva dela. Zgoraj so zavihki s seznami algoritmov, testnih množic, vhodnih in izhodnih parametrov in vsega ostalega kot v uporabniškem vmesniku za poizvedbe v formatu JSON. Spodaj je poizvedba v tekstualni obliki. Če zgoraj kliknemo na nek gumb, se ustrezen element zapiše v spodnje besedilo poizvedbe. Besedilo lahko tudi direktno urejamo. Možno je tudi oboje hkrati, z miško lahko klikamo po poljih zgoraj, s tipkovnico pa vnašamo ostale znake. To je še posebej uporabno pri vnosu izračunanih polj, ko lahko polja vnesemo s klikom, operande in ostale znake pa s tipkovnico. Ob kliku na gumb Run! se poizvedba izvede v glavnem oknu uporabniškega vmesnika.

3.2.2 Izračunana polja

V izhodna polja je bila dodana možnost izračunanih polj. Izračunana polja delujejo tako, da poleg izhodnih polj, ki so predvidena v definiciji projekta, lahko dodamo še dodatna polja, izpeljana iz obstoječih.

V izračunano polje lahko dodamo poljuben izraz. Rezultat je lahko niz, število ali logična vrednost. Vrednost izraza se izračuna s pomočjo odprtokodnega interpreterja programske kode Java z imenom BeanShell. BeanShell omogoča dinamično izvajanje kode napisane v Javi, podpira pa tudi elemente skriptnih jezikov. Tako na primer v kodi ni potrebno določiti tipa spremenljivke. To nam je prišlo prav pri izračunu izraza v izračunanih poljih. Interpreterju damo za izračunati izraz »`result = izraz izračunanega polja`«, brez da bi določili tip spremenljivki `result`. Ta je tako lahko niz, število ali logična vrednost.

V izraz izračunanega polja lahko vstavimo poljubne vhodne ali izhodne parametre, konstante in funkcije, ki so na voljo v programskem jeziku Java. Funkcije razreda Math so še bolj enostavne za uporabo, ker jim ni potrebno določiti razreda, ampak se ta doda avtomatsko. Tako lahko na primer zapišemo `abs(Tmin)` namesto `Math.abs(Tmin)`. Vrednost izračunanega polja se izračuna za vsak algoritem, ki je zajet v poizvedbi. Izhodnim parametrom uporabljenim v izrazu lahko določimo algoritem. Na ta način imamo v istem izrazu lahko polja večih algoritmov.

3.2.2.1 Filtriranje izračunanih polj

Nad izračunanim poljem je možno določiti filter. V filtru je možno uporabiti izračunano polje samo, če je to izračunano polje tudi med izhodnimi polji. Če je v izračunanem polju definiran algoritem, mora biti enako definiran tudi v filtru. Lahko je pa v filtru dodatno definiran algoritem, tudi če ni v izračunanem polju.

Primer:

Imamo dva algoritma: QuickSort in BubbleSort. Za izhodna polja določimo `Tmax` in `Tmin`, dodatno pa še izračunano polje `Tmax-Tmin`.

Definiramo različne filtre nad izračunanim poljem:

- `QuickSort.Tmax-Tmin>0`: algoritem je definiran za prvi operand. Filter lahko razumemo kot `QuickSort.Tmax-QuickSort.Tmin>0 IN QuickSort.Tmax-BubbleSort.Tmin>0`.

- `Tmax-QuickSort.Tmin>0`: algoritem je definiran za drugi operand. Filter lahko razumemo kot `QuickSort.Tmax-QuickSort.Tmin>0 IN BubbleSort.Tmax-QuickSort.Tmin>0`.

- `QuickSort.Tmax-QuickSort.Tmin>0`: filter deluje samo nad enim točno določenim poljem.

3.2.2.2 Opis delovanja

Izračunana polja se v formatu JSON zapišejo med ostala izhodna polja, tako da spremembe formata poizvedbe niso bile potrebne. Spremeniti pa je bilo potrebno glavno funkcijo, ki izvaja poizvedbe. Najprej je bilo potrebno programsko dodati še dodatna izhodna polja, za vsak algoritem in za vsako izračunano polje po enega. Pri izračunu operandov izračunanega polja se je uporabil že obstoječ izračun običajnega izhodnega parametra. Tako se za vsak algoritem izraz izračunanega polja preoblikuje tako, da se namesto parametrov v izrazu vstavijo konkretne vrednosti, nov izraz se pošlje v interpreter programske kode BeanShell, rezultat interpreterja pa se zapiše kot rezultat polja v tabelo.

Pri filtriranju po izračunanem polju je bilo potrebno spremeniti samo funkcijo, ki iz izraza filtra izračuna polja, na katera se filter nanaša.

Za grupiranje in sortiranje po izračunanem polju niso bile potrebne dodatne spremembe.

3.2.3 Optimizacija

V Algatorju so se nekatere poizvedbe izvajale preveč časa. Še posebej je bil problem pri filtru s parametrom, ki deluje tako, da filtru nastavimo spodnjo in zgornjo mejo ter korak. Poizvedba se potem izvede za vsako vrednost filtra in združeno pokaže v rezultatu. Ob tem mora biti obvezno vklopljena možnost štetja vrstic. Problem je bil v tem, da se je poizvedba na novo izvedla v celoti za vsak korak filtra. Tudi nekatere druge poizvedbe so trajale predolgo.

V prvotni verziji se poizvedba izvede po naslednjem postopku:

1. Iz poizvedbe se prebere kateri algoritmi in testne množice bodo zajete v poizvedbi.

2. Za vsak algoritem in za vsako testno množico se iz datoteke z opisom projekta prebere vrstni red vhodnih in izhodnih parametrov, ter imena vhodnih in izhodnih parametrov.

3. Za vsak algoritem in za vsako testno množico se ustvari tabela, zaenkrat s samo definiranimi stolpci, ki bodo zajeti v rezultatu. Stolpci vedno vsebujejo ime algoritma, ime testne množice, oznaka testne množice, status izvedbe testa. Sledijo še vhodni in izhodni parametri.

4. Za vsak algoritem in za vsako testno množico se iz datotek z rezultati testa preberejo vse vrstice. Za vsako vrstico datoteke se doda vrstica v tabelo ustvarjeno v prejšnjem koraku z vrednostmi stolpcev kot se preberejo iz datoteke.

5. Iz poizvedbe se preberejo vhodni in izhodni parametri.

6. Začne se sestavljati tabela, ki se bo izpisala kot rezultat. Najprej se v glavo tabele zapišejo imena stolpcev. Tabela zmeraj vsebuje številko vrstice, ime testne množice, oznako testne množice in status.

7. V glavo tabele se dodajo vhodni parametri, ki so navedeni v poizvedbi.

8. V glavo tabele se dodajo vsi izhodni parametri za vsak algoritem posebej, v obliki `<ime algoritma>.<ime parametra>`.

9. Napolni se vsebinski del tabele. Za vsako vrstico iz vmesne tabele z rezultati se doda vrstica v končno tabelo.

10. Tabela se filtrira glede na filter prebran iz poizvedbe. To se naredi tako, da se za vsak filter za vsako vrstico izračuna vrednost izraza v filtru. Če vrstica ne ustreza filtru, se izloči.

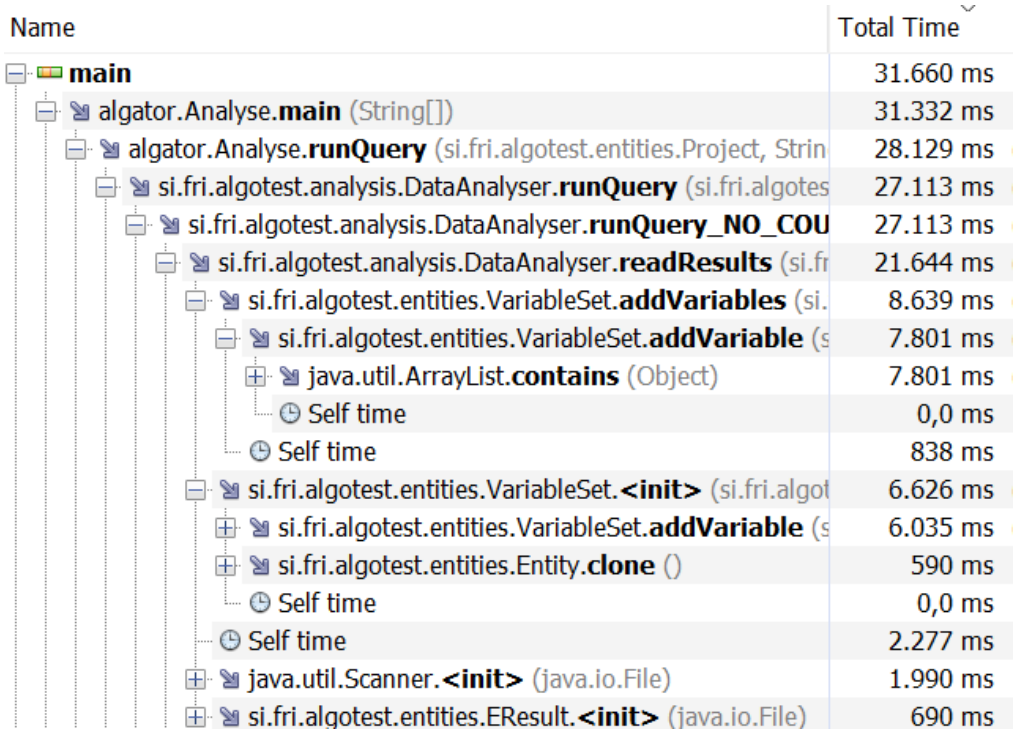
11. Če poizvedba vsebuje polja za grupiranje, se grupiranja izvede tako, da se za grupirano polje prikažejo samo vrstice z različnimi vrednostmi, ostala polja pa vsebujejo prvo vrednost.

12. Sledi še sortiranje tabele po poljih kot so navedena v poizvedbi.

13. Če je vklopljeno štetje vrstic se ves postopek ponovi za vsako vrednost filtra in za vsak algoritem. Končna tabela z rezultati v tem primeru vsebuje samo stolpce z imenom algoritma.

14. Tabela se izpiše.

Optimizacije smo se lotili tako, da smo najprej zagnali orodje, ki nam prikaže koliko časa se izvaja katera funkcija v programu. V programu NetBeans se imenuje Profiler. Orodje smo zagnali nad sicer enostavno poizvedbo brez filtriranja, ampak z veliko podatki. Dobili smo seznam, ki je prikazan na sliki 3.7.

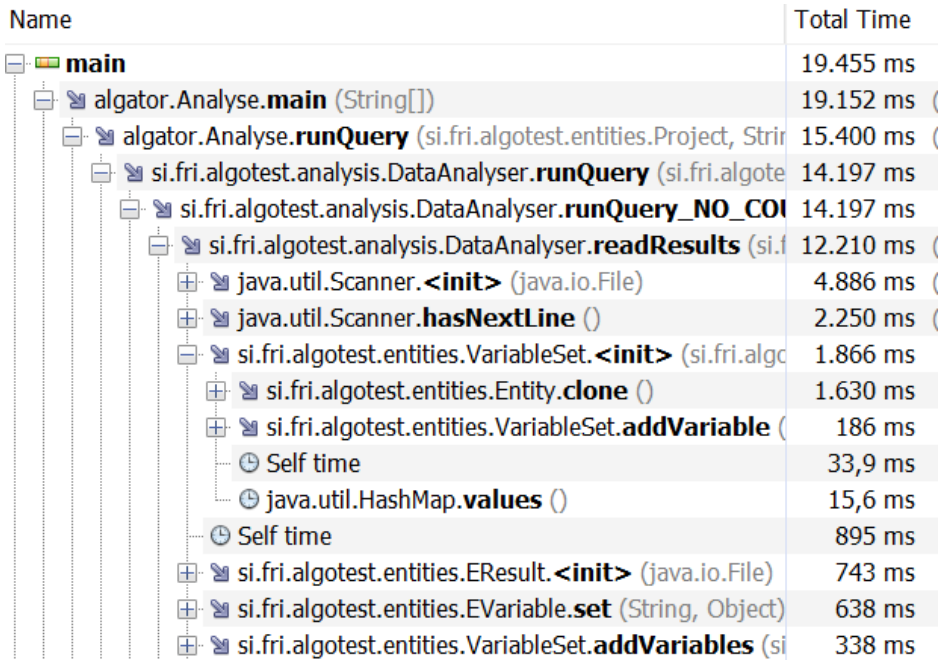


Slika 3.7: Rezultat programa Profiler za poizvedbo z neoptimiziranim Algatorjem.

Vidimo, da največ časa vzame metoda z imenom `addVariables`, ki v zanki kliče funkcijo `addVariable`. Naslednja metoda na seznamu je konstruktor razreda VariableSet, znotraj tega pa spet daleč največ časa vzame metoda `addVariable`. Znotraj metode `addVariable` pa vidimo, da praktično ves čas vzame klic metode contains razreda `ArrayList`.

Pri pregledu metode `addVariable` smo ugotovili tako, da sprejme dva parametra; parameter `variable` tipa `EVariable`, ki se naj doda v seznam, ter parameter `replaceValue`, ki pove ali se naj vrednost nadomesti, če že obstaja v seznamu. Metoda v

ozadju uporablja seznam `ArrayList`. Najprej se sprehodi čez celoten seznam, da preveri ali je parameter variable že v seznamu. Če ni, ga doda v seznam. Če pa je, se preveri parameter `replaceValue`, in če je v njem vrednost `true`, se vrednost elementa v seznamu nadomesti z novo vrednostjo, sicer ne naredi ničesar. Problem te metode je v tem, da se v vsakem primeru pregleda celoten seznam, oz. vsaj do tistega elementa, ki je enak parametru variable. Če je že v seznamu, se še enkrat pregleda seznam, da poišče ustrezen element v seznamu, ki se mu spremeni vrednost. Očitno je problem v iskanju ustreznega elementa. Pregledali smo razred `EVariable`, ki mu pripada prvi parameter. Vsebuje metodo equals, ki vrne enakost dveh primerkov tipa `EVariable` v primeru, da imata enako vrednost funkcije `getName()`. To lastnost smo uporabili pri spremenjenem iskanju elementa v seznamu. Tip polja variables smo spremenili iz `ArrayList` v `HashMap<String, EVariable>`. Prvo polje elementa v seznamu napolnimo z vrednostjo, ki jo vrne `getName()`, drugo pa s samim primerkom razreda `EVariable`. Zaradi te spremembe je bilo potrebnih še nekaj drugih popravkov znotraj razreda `VariableSet`. Funkcija `getVariable(int i)` v razredu `VariableSet` je postala neučinkovita, ker iskanje po zaporedni številki v razredu `HashMap` ni podprto, in je potrebno seznam vrednosti iz `HashMap` najprej spremeniti v kakšen drug seznam, ki podpira takšno iskanje. Pri pregledu programa Algator smo ugotovili, da se ta funkcija uporablja samo znotraj nekaj zank s števci, ki se jih bi dalo enostavno spremeniti, da ne bi uporabljale števca, ampak elemente seznama. Za to je bilo potrebno razredu `VariableSet` dodati implementacijo vmesnika `Iterable<EVariable>`. Tako je postalo možno razred `VariableSet` uporabiti kot vir elementov v zanki for. Še enkrat smo pognali Profiler in dobili rezultat na sliki 3.8.



Slika 3.8: Rezultat programa Profiler z optimizirano funkcijo `addVariable`.

Vidimo, da metoda `addVariable` znotraj konstruktorja razreda `VariableSet` traja 186 ms namesto 6035 ms, `addVariables` v metodi `readResults` pa traja 338 ms namesto 8639 ms. Skupni čas izvajanja se je zmanjšal iz 31 sekund na 19.

Takšen rezultat smo dobili samo s spremembo enega manjšega dela programa. Iz postopka poteka izvedbe poizvedbe pa se vidi, da je tudi znotraj njega prostor za izboljšave:

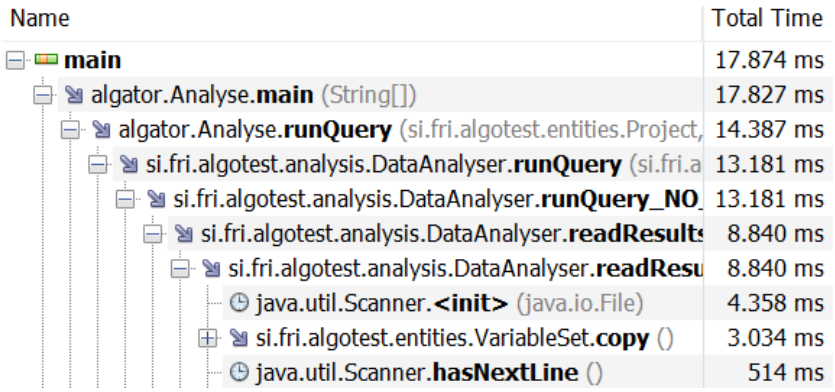
- V 2. koraku se podatki o projektu prebirajo iz datoteke za vsak algoritem in za vsako testno množico, čeprav so vsi algoritmi in vse testne množice znotraj istega projekta.

- Podobno se v 3. koraku za vsak algoritem in za vsako testno množico na novo ustvarja tabela, čeprav so na tej točki vse tabele enake.

Ta dva problema smo popravili s predpomnjenjem vmesnih tabel. Predpomnilnik je implementiran v razredu `Cache`, ki v ozadju uporablja `HashMap<Object, CacheEntry>`. Ključ je torej tipa `Object`, `CacheEntry` je pa nov razred, ki vsebuje predpomnjeno vrednost ter čas zadnjega dostopa. Razred Cache ima dve funkciji, set in get. Set sprejme dva parametra, ključ pod katerim se naj shrani podatek in sam podatek. Ob zapisu podatka se shrani še trenutni čas. Funkcija get sprejme en parameter, in sicer ključ. Če obstaja zapis v predpomnilniku s tem ključem, metoda vrne podatek ter zapiše nov čas zadnjega

dostopa v predpomnilnik. Če zapis s tem ključem ne obstaja, metoda vrne `null`. Ob vsakem zagonu poizvedbe se preveri ali je predpomnilnik še veljaven, in če ni, se celoten izbriše. To naredi tako, da preveri ali je bila kakšna datoteka z rezultati testov nad testnimi množicami spremenjena od zadnjega zagona poizvedbe. Posamezni zapisi v predpomnilniku se izbrišejo, če določen čas niso bili prebrani. Razred `Cache` je narejen splošno in ga je mogoče uporabiti kjerkoli v programu Algator.

V predpomnilnik smo dodali še tabelo z vsemi rezultati pred filtriranjem. Pri izvedbi samo ene poizvedbe sicer od tega ni nobene koristi, čas poizvedbe se še poveča, ker je treba tabelo kopirati v predpomnilnik. Izkazalo se je, da je ta povečan čas zaradi kopiranja te tabele približno tolikšen kot je prihranek pri pomnjenju prejšnjih vmesnih podatkov (slika 3.9), ki se ne berejo več za vsak algoritem in testno množico posebej. Veliko pa se prihrani na času pri večih zaporednih poizvedbah, ki imajo enak izbor parametrov in samo različen filter, grupiranje ali sortiranje. V tem primeru se preskočijo vsi koraki do filtriranja, takšna poizvedba pa traja zanemarljivo časa. Več zaporednih poizvedb z enakimi parametri in s samo različnimi filtri lahko poganjamo preko uporabniškega vmesnika ali pa kot filter s parametrom.



Slika 3.9: Nova metoda na seznamu je `VariableSet.Copy`, ki se uporablja za kopiranje vmesnih tabel v predpomnilnik. Skupni čas prvega zagona poizvedbe se ni veliko zmanjšal, so pa zdaj podatki pripravljeni za naslednjo podobno poizvedbo, ki se bo izvedla veliko hitreje.

Po končani optimizaciji filter s parametrom deluje veliko hitreje, uporabniški vmesnik je pa bolj odziven, posebej če izvajamo več podobnih poizvedb eno za drugo.

Poglavje 4 Sklepne ugotovitve

Primerjali smo poizvedovalne jezike SQL, XQuery in LINQ po različnih lastnostih. Ugotovili smo, da so si podobni po ekspresivnosti, imajo pa tudi vsak svoje lastnosti in funkcije, ki ustrezajo njihovemu namenu. SQL je namenjen predvsem delu z relacijskimi podatkovnimi bazami in na tem področju ima največ funkcionalnosti. XQuery je namenjen poizvedovanju po dokumentih XML. Nima vseh mehanizmov kot SQL, je pa mogoče z njim enostavno poizvedovati nad dokumenti XML, pa tudi rezultate predstaviti v isti obliki. LINQ na nek način združuje prednosti obeh jezikov, obenem se pa uporablja v splošnem programskem jeziku, ki omogoča izdelavo celotne aplikacije, in ne samo podatkovnega dela. LINQ se lahko dobro poveže s podatkovno bazo SQL, ne da se pa preko njega uporabljati vseh njenih funkcionalnosti. Za vir podatkov lahko uporabi tudi dokumente XML. Tudi v tem primeru ne zmore uporabiti čisto vseh podatkov, ki jih dokument XML lahko vsebuje. Za vse tri jezike velja, da omogočajo veliko razširitev, s katerimi jih po funkcionalnosti lahko tudi izenačimo. Te razširitve so odvisne predvsem od posamezne implementacije jezika.

Določili smo sintakso za poizvedovalni jezik za uporabo v sistemu Algator. Sintaksa je podobna jeziku SQL, je pa poenostavljena, ker nekateri elementi niso potrebni. Nov poizvedovalni jezik smo vgradili v sistem tako, da je njegovo izvajanje ostalo čim bolj enako prvotnemu. Prvotna izvedba ni bila optimalna, zato so se nekatere poizvedbe izvajale predolgo. S pomočjo programa Profiler smo poiskali katere metode se v programu izvajajo najdlje časa in jih optimizirali. Dodali smo še funkcionalnost izračunljivih polj, ki deluje tako, da izraz pošlje v interpreter programske kode Java, ki izračuna vrednost izraza in ga vrne kot rezultat. Interpreter bi lahko uporabili še za kakšno drugo funkcionalnost, npr. za izračun vrednosti celotnega filtra, ker zna računati tudi z logičnimi vrednostmi.

Literatura

[1] Charlie Calvert in Dinesh Kulkarni, "Essential LINQ": Addison-Wesley, 2009.

[2] Joe Celko, "SQL for smarites: advanced SQL programming: Morgan Kaufman Publishers, 2005.

[3] Tomaž Dobravec : Algator [Online]. Dosegljivo: https://github.com/ALGatorDevel/Algator.

[4] Ramez Elmasri in Shamkant B. Navathe, "Fundamentals of Database Systems": Addison-Wesley, 2011.

[5] Peter Haase, Jeen Broekstra, Andreas Eberhart, Raphael Volz, "A Comparision of RDF Query Languages", University of Karlsruhe, Vrije Universiteit Amsterdam, 2004.

[6] Jim Melton in Stephen Buxton, "Querying XML": Morgan Kaufman Publishers, 2006.

[7] Steven van Ombergen: "A Comparison of Five Document-Store Query Languages": University of Amsterdam, 2014.

[8] Priscilla Walmsley, "XQuery": O'Reilly Media, 2007.

[9] Introduction to LINQ Queries (C#) [Online]. Dosegljivo: https://msdn.microsoft.com/en-us/library/bb397906.aspx

[10] C# Features that support LINQ [Online]. Dosegljivo: https://msdn.microsoft.com/en-us/library/bb397909.aspx

[11] XQuery 3.0: An XML Query Language [Online]. Dosegljivo: http://www.w3.org/TR/xquery-30.

[12] XQuery Update Facility 1.0 Requirements [Online]. Dosegljivo: https://www.w3.org/TR/xquery-update-10-requirements/